MASTER EN SCIENCES INFORMATIQUES

Manipulation de versions multiples dans un atelier logiciel intègre

Castaigne, Claude; Dinsart, Alain

Award date:
1984

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX

INSTITUT D'INFORMATIQUE

NAMUR

MANIPULATION DE VERSIONS MULTIPLES DANS UN ATELIER LOGICIEL INTEGRE.

Claude CASTAIGNE.

Alain DINSART.

Promoteur : A. van Lamsweerde

Mémoire présenté en vue de l'obtention du grade de Licencié et Maître en Informatique.

Année Académique 1983 - 1984.

Avant toute chose, permettez-nous d'effectuer ici quelques remerciements aux personnes sans lesquelles ce travail n'aurait jamais pu arriver à bonne fin.

Tout particulièrement, nous nous devons de remercier Monsieur A. van Lamsweerde, promoteur de ce mémoire, pour l'intérêt qu'il a manifesté tout au long de l'élaboration de ce travail. Son expérience lui a permis de nous donner de judicieux conseils sans lesquels ce mémoire ne serait pas ce qu'il est aujourd'hui.

Nous voudrions également exprimer toute notre gratitude à l'équipe Adèle du centre informatique IMAG de l'Université de Grenoble, pour son acceuil, sa gentillesse, qui nous ont permis de joindre l'utile à l'agréable.
Nous remercions tout particulièrement Jacky Estublier pour sa disponibilité, ses idées et sa direction éclairée.

Nous témoignons aussi toute notre gratitude à Dominique qui a su sacrifier une grande partie de ses loisirs pour dactylographier ce travail.

Enfin, nous ne pouvons oublier toutes ces personnes de l'ombre, qui de près ou de loin, nous ont aidés à réaliser ce mémoire. Merci.

TABLE DES MATIERES.

R.C. : Rédaction Commune.
C.C. : Claude Castaigne.
A.D. : Alain Dinsart.

INTRODUCTION.

Il devient maintenant traditionnel d'introduire tout travail lié au génie logiciel en rappelant quelques chiffres significatifs. Selon (Kra, 82) ou (Boe, 82), le coût du logiciel occupe une part croissante du coût total d'un système informatique. Les coûts de maintenance deviennent prohibitifs et peuvent atteindre jusuqu'à 70 % du prix du logiciel. Quant au développement, la part d'écriture ne représente qu'environ 20 %, pour environ 30 % d'analyse et de conception, et 50 % de mise au point, test et intégration.

Les coûts élevés de maintenance résultent surtout d'un manque de qualité du produit développé, mesurée en terme d'adéquation aux besoins, de fiabilité, de performance, de transférabilité et d'adaptibilité à des besoins qui évoluent (Lam, 82).

Cette situation communément connue sous le vocable de "crise du software" a incité des milieux informatiques à développer des systèmes d'assistance à la production du logiciel, encore appelés environnements ou ateliers logiciels.

Ce processus complexe de production se subdivise en plusieurs étapes pouvant bénéficier chacune d'outils d'aide à sa réalisation. Rappelons que le cycle de vie d'un logiciel s'articule autour des phases suivantes: analyse des besoins, spécifications, conception, programmation et codage, test et intégration, exploitation et maintenance. Historiquement, on s'est surtout intéressé à la phase de codage en développant des assembleurs, des compilateurs ou encore des éditeurs de texte. Mais par la prise de conscience de l'existence du cycle de vie du logiciel, des besoins d'assistance se sont avérés tout aussi substanciels pour chacune des autres étapes.

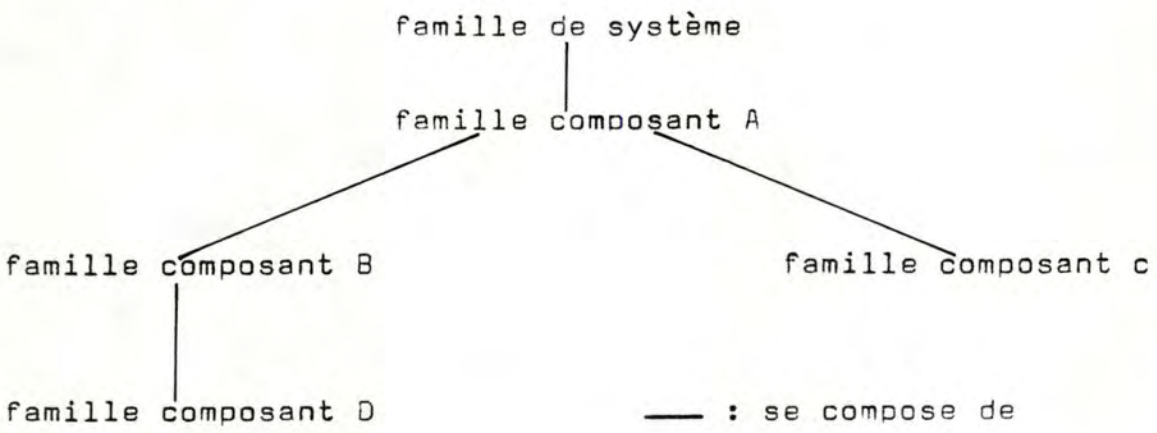
Une spécification mal ordonnée peut s'avérer néfaste pour les phases ultérieures. On trouvera dans (Lam, 82) un état de l'art et un aperçu des tendances actuelles concernant les systèmes d'aide au développement de logiciel supportant cha-

cune des phases de son cycle de vie, et dans (Dro, 82), une présentation des principaux développements français actuels.

La phase qui nous intéresse plus particulièrement est celle du codage ou de la programmation. Le but essentiel des outils supportant cette étape vise à l'amélioration de la productivité du programmeur. Cet objectif nécessite la réduction du cycle compilation-test-correction-recompilation en limitant les risques d'erreurs. C'est ainsi que l'on assiste actuellement à une poussée des éditeurs syntaxiques et formateurs, interpréteurs et metteurs au point. Un aspect non négligeable lié à la productivité réside dans la qualité du poste de travail. On constate à cet égard la mise au point d'interfaces usagers sophistiquées permettant le multi-fenêtrage et réduisant les limites physiques des écrans en adoptant des stratégies d'écrans virtuels.
La tendance actuelle consiste à intégrer ces outils dans un ensemble unique et cohérent. On parle dès lors d'ateliers intégrés de logiciel. Selon (Lam, 82), cette intégration ne requiert pas nécessairement une structure complexe d'outils interconnectés, mais peut reposer sur l'utilisation d'une structure de données commune. Dans le cadre de codage qui nous occupe, la structure de plus en plus reconnue est la représentation abstraite des programmes sous une forme arborescente, d'autant plus malléable qu'elle reflète avec précision la structure du texte qu'elle représente. Cette structure, ou représentation interne, nécessite le développement de décompilateurs permettant de transformer la forme interne arborescente en une représentation externe textuelle, et d'introducteurs ou analyseurs possédant la fonction inverse. Elle induit en outre le développement d'un gestionnaire d'arbres offrant les primitives nécessaires à la manipulation de la structure.

Outre ces quelques outils essentiellement axés vers l'aide au codage, on constate actuellement la nécessité de dévelop-

per des bases de données généralisées, c'est-à-dire non limitées dans la structure des informations qu'elles contiennent. De telles bases sont exploitées pour conserver les programmes et gérer les différents types de relation qui les associent avec des entités aussi variées que leur documentation, spécification, jeux de tests, codes objets... D'autre part, il s'avère nécessaire que tout produit logiciel existe en versions multiples. Que ce soit dans les systèmes d'exploitation ou d'informations, les besoins évoluent et pour faire face à cette évolution, on passe le plus souvent de la version courante à une version plus adaptée. On aboutit ainsi à une "famille" de systèmes rencontrant les besoins d'une "famille" d'utilisateurs suffisamment proches. Un problème aigu lors du passage d'une version à la suivante consiste à pouvoir localiser les éléments à modifier et à focaliser l'intervention au niveau de ces seuls éléments. Cela ne sera possible que si le système a été subdivisé en composants d'une façon suffisamment souple qu'il ne soit pas nécessaire d'agir tous azimuts pour satisfaire les nouveaux besoins. Ainsi le passage d'une version à la suivante s'opérera par la modification des seuls composants affectés et on assistera à la création de "familles" de composants.
Une famille de systèmes se décomposera par conséquent de la façon suivante:



où toute famille de composants comprend au moins un élément

et où tout composant englobe aussi bien les notions de programmes ou codes objets, que spécification, documentation, jeux de tests...

Si nous n'avons argumenté la nécessité de gérer des versions multiples que par l'évolution des besoins, il n'en est pas moins vrai que de telles versions peuvent naître en phase de développement par le biais d'essais de nouveaux algorithmes, ou en phase d'exploitation par la nécessité d'adapter le logiciel à des configurations multiples de matériel.
Quelle qu'en soit l'origine, la base de données doit pouvoir gérer de telles structures. Un problème inhérent à l'exploitation de versions multiples de systèmes consiste à pouvoir régénérer un système particulier en sélectionnant l'ensemble des composants appropriés. Un problème inhérent à la conservation de cette multiplicité consiste à éviter de gaspiller une quantité trop importante de place mémoire. Il est en effet fréquent que deux ou plusieurs composants d'une même famille soient suffisamment proches par leur contenu et il est intéressant dans une telle situation de réduire leur conservation aux seules différences qui les séparent.

PLAN DU MEMOIRE.

Le présent mémoire se subdivise en deux parties. La première relate le cadre existant. Pour commencer, nous présentons l'environnement dans lequel nous avons travaillé, c'est-à-dire le projet Adèle. Ce projet, un atelier de logiciel intégré, est divisé en quatre sous-projets à savoir une interface-usager, un système automatique de production de générateur de code, une représentation interne de programmes et une base de programmes. Ces sous-projets sont développés indépendamment les uns des autres. Nous ne donnons qu'un bref aperçu des trois premiers, étant donné que notre travail se focalisait surtout sur le quatrième. Celui-ci

fait l'objet d'une description beaucoup plus complète.

Dans un second temps, nous abordons le système RCS. Celui-ci est en fait un système de gestion de révisions. A sa base se situe toute une série de concepts que nous énumérons en parallèle à ceux utilisés pour développer notre travail.

La deuxième grande partie de ce mémoire présente la réalisation et l'intégration d'une gestion d'historiques et des fonctions de mémorisation de versions dans la base Adèle.

Un premier chapitre est la gestion des historiques. Cet objet se charge de retracer l'évolution chronologique de tout objet se trouvant dans la base. C'est ainsi que dans un premier temps, nous donnons les objectifs et les fonctionnalités d'un tel système. Dans un second temps, nous abordons le problème de sa réalisation pratique et nous explicitons les modules qui l'entourent.
Pour être complets, nous ne pouvons parler d'historiques sans évoquer brièvement les problèmes rencontrés lors de l'intégration dans la base et les choix d'implémentation qui en résultent.

Le second chapitre est consacré plus spécialement à la mémorisation de programmes développés en versions multiples. Un premier point présente nos objectifs et les fonctionnalités de notre système de mémorisation. Nous nous attardons ensuite aux alternatives de conception et de réalisation en explicitant nos différents choix.
Après avoir abordé l'architecture de la découpe en modules de ce système, nous les décrivons, spécifions et en donnons les choix de réalisation.

Le troisième chapitre de cette seconde partie relate les aspects liés à l'intégration des deux outils réalisés dans

la base Adèle.

Pour en terminer avec cette deuxième et dernière partie, nous tentons dans un quatrième chapitre de présenter les avantages, les inconvénients et les limites de nos deux travaux, et nous étudions dans un ultime chapitre une possibilité d'extension qui consiste à mémoriser les programmes sous une forme arborescente. Nous y étudions plus particulièrement l'incidence qu'une telle mémorisation aurait eue sur notre second travail.

En guise de conclusion à ce mémoire, nous avons jugé intéressant de soumettre à la critique du lecteur nos différents points de vue sur l'ensemble du travail réalisé ainsi que sur son contexte et son environnement.

PREMIERE PARTIE : le cadre existant.

1. ADELE : un atelier de développement logiciel.

1.1. Introduction.

L'étude dont il est question dans ce mémoire a été réalisée dans le cadre du projet ADELE (Atelier de DEveloppement de LogiciEl) au laboratoire IMAG à Grenoble.
Ce projet a pour but d'étudier divers aspects de la conception et de la réalisation d'ateliers intégrés de production de logiciel et de valider cette étude par la réalisation d'un prototype expérimental.
Les outils considérés jusqu'à présent se réfèrent aux phases de programmation (conception de programmes, leur codage, et leur mise au point) et de maintenance (gestion de versions multiples).
Le langage source pour rédiger les programmes sous l'atelier est unique: il s'agit de Pascal SOL, une extension de Pascal permettant de représenter des modules. Le prototype expérimental est implémenté sur une configuration HB68 sous le système d'exploitation MULTICS.

A notre arrivée, ce prototype comprenait un éditeur syntaxique inspiré du "The Cornell Program Synthetizer" (Tei, 81) et de l'éditeur syntaxique de Mentor (Don, 79), et un outil de mise au point construit autour d'un interpréteur.
De plus, les programmes développés sont mémorisés dans une base de données centralisée, sous une forme interne bien adaptée à leur édition et à leur interprétation. Le squelette de cette représentation interne (RI) est un arbre syntaxique de programme décoré de quelques attributs. Cette forme intermédiaire sert également de point de départ à une génération de code paramétrée.

De plus, deux outils sont associés à cette RI:

- un introducteur permettant l'entrée dans l'atelier de programmes Pascal existants déjà ;
- un décompilateur permettant de reconstituer un programme Pascal Standard.

Un système est composé de plusieurs modules. De par l'évolution des besoins, les modules d'un logiciel sont amenés à coexister en plusieurs versions dans la base de données. Il est donc nécessaire de définir:

- une technique permettant de sélectionner les bonnes versions de modules pour un logiciel particulier ;
- des règles de cohérence entre les versions multiples d'un même module. Une règle peut s'énoncer comme suit: tout logiciel se trouve dans un état incohérent si pour au moins un de ses modules, les techniques de sélection ne permettent pas de choisir une version d'un module qui convienne.

Afin d'améliorer la qualité du poste de travail, un interface-usager a été créé, permettant une utilisation aisée des outils décrits plus haut.

En résumé, l'équipe Adèle étudie les quatre aspects suivants:

- la représentation interne des programmes sous une forme intermédiaire, construite autour du concept d'arbre abstrait syntaxique, et les outils de manipulation de cette représentation ;
- la production automatique de générateurs de code à partir d'une description de la machine cible. Cette description contient d'une part la structure de la machine (mémoires, registres ...) et d'autre part le jeu d'instruction de celle-ci ;

- les principes de conception de l'interface du poste de travail: gestion des fenêtres, édition d'interfaces et affichage des informations ;
- un système d'archivage permettant la gestion des versions d'un module.

L'architecture générale de l'atelier peut se dessiner comme suit:

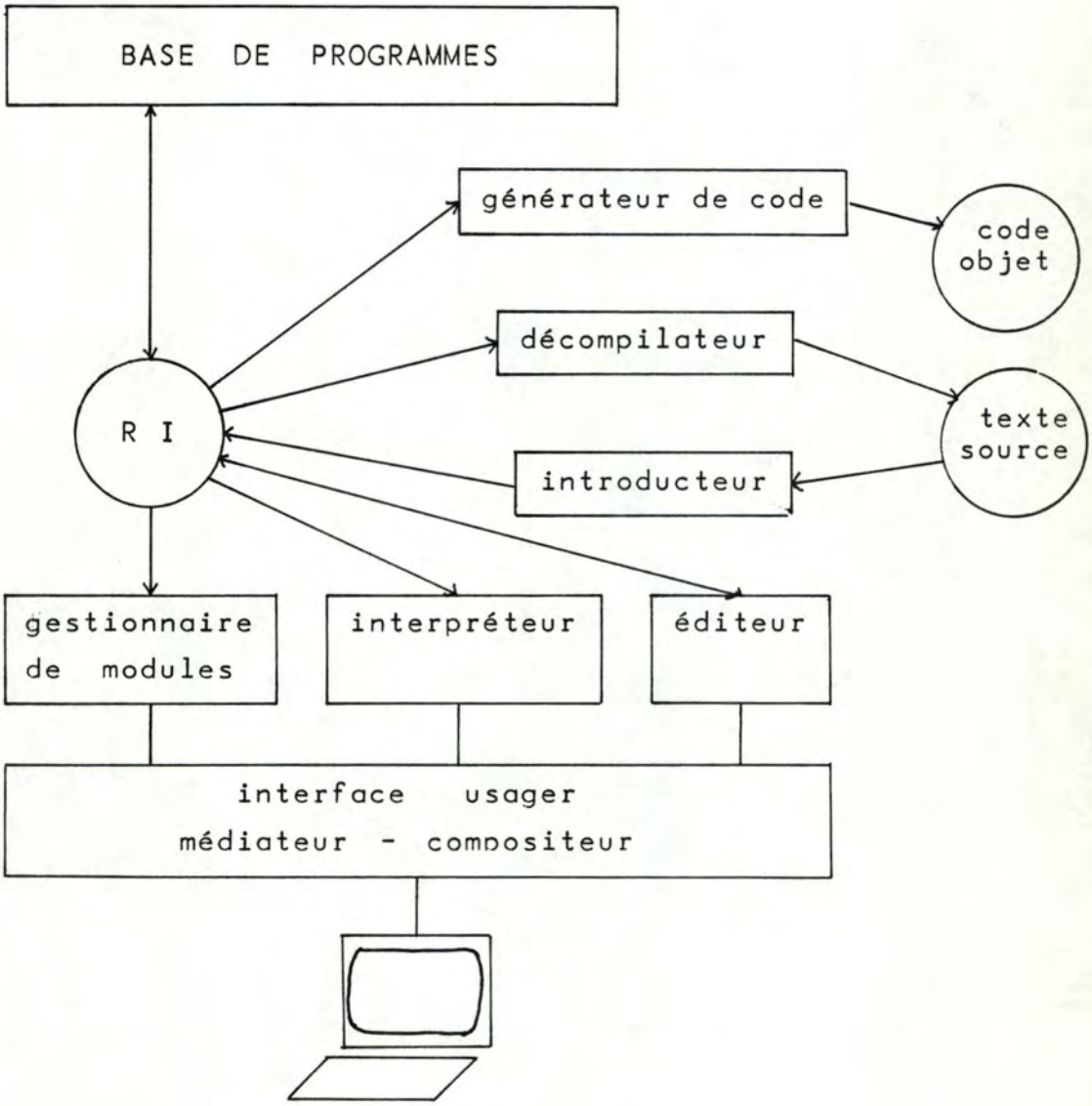


fig.1.1 Architecture générale.

1.2. Concepts généraux.

Nous présentons ici les quatre aspects étudiés par l'équipe Adèle: la représentation interne, le générateur de code, l'interface-usager, et la base de programmes. C'est à l'intérieur de cette dernière que nous avons travaillé et nous la décrirons plus précisément à la section 1.3. . Comme les trois autres aspects n'intervenaient pas dans le cadre de notre travail, nous n'en donnons qu'un bref aperçu. Le lecteur intéressé par plus de détails pourra se reporter à la bibliographie Adèle.

1.2.1. La représentation interne.

Considérons les étapes classiques lorsqu'on introduit un programme en vue de l'exploiter:

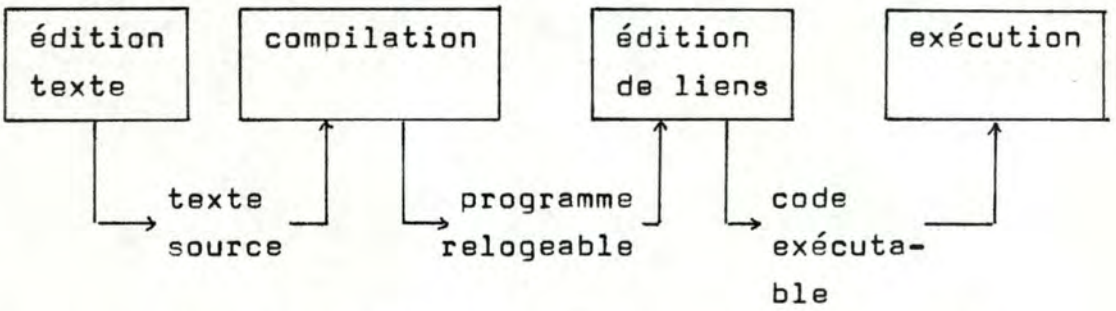


fig.1.2 Cycle de développement d'un programme.

Les erreurs syntaxiques sont généralement détectées lors des phases de compilation et d'édition de liens. Leur correction implique alors un retour à la phase d'édition de texte, au cours de laquelle le programme sera modifié, puis à la phase de compilation. De plus, les erreurs syntaxiques découvertes par le compilateur sont pour la plupart des erreurs triviales (omission d'une parenthèse, erreur de frappe d'un nom de variable ...). Leur correction ne demande en général que quelques commandes de l'éditeur et il est souvent plus long de se positionner sur l'erreur que de la corriger. Il en est de même pour les erreurs détectées à

l'exécution, avec cette fois une augmentation du nombre de phases à recommencer.
Afin de raccourcir ce processus, on peut utiliser un éditeur syntaxique (incluant certaines fonctions d'analyse grammaticale du compilateur) et un interpréteur du langage (cfr. supra). Théoriquement, on peut ainsi éditer, mémoriser et interpréter des textes sources.
Pour des raisons d'efficacité, on mémorise les programmes sous forme d'une représentation intermédiaire basée sur l'arbre syntaxique du programme et que nous décrirons plus tard. La structure et le contenu de cette représentation semblent plus adaptés à l'édition syntaxique du programme et à l'interprétation. En effet, un éditeur syntaxique a besoin de trouver facilement les composants d'une instruction if par exemple (condition, partie then, partie else). La structure qui semble la mieux adaptée à ce type de tâche est l'arbre syntaxique car il reflète directement les structures du langage.

Il en résulte le nouveau schéma suivant:

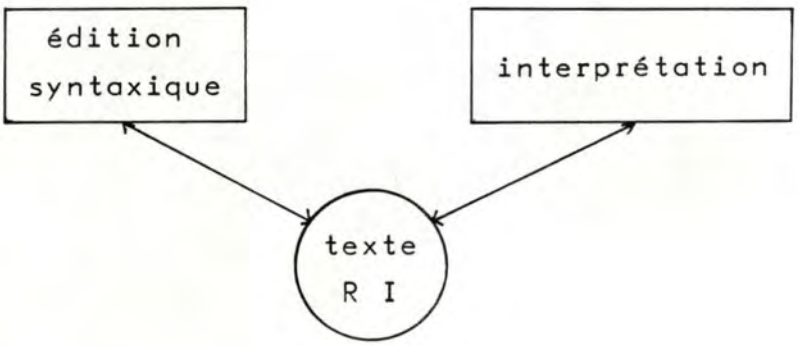


fig.1.3 La RI.

La représentation interne d'un programme constitue la forme unique sous laquelle celui-ci sera manipulé lors de l'édition, l'interprétation et la génération de code. En conséquence, elle doit avoir les propriétes suivantes:

- toutes les informations relatives au programme source se retouvent dans sa RI :
- elle est bien adaptée pour être traitée par les outils de l'environnement et plus spécialement par l'éditeur ;
- elle contient des informations syntaxiques (pour l'édition syntaxique) mais également sémantiques (pour l'interprétation et la génération de code).

La structure de base de la RI est un arbre abstrait du programme source. A chaque unité syntaxique du programme correspond un noeud dans la RI.
Par exemple, l'intruction conditionnelle if C then I1 else I2, est représentée par un noeud "if trt" avec trois branches correspondant à C, I1 et I2.

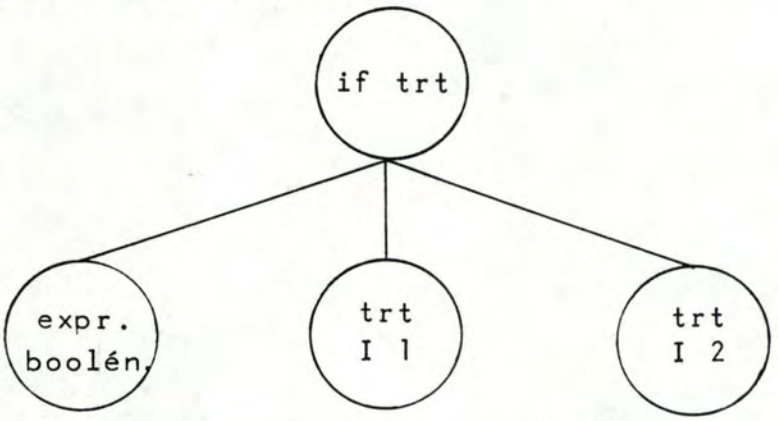


fig.1.4 Arbre syntaxique.

Un noeud possède différents attributs:

- les attributs de structure caractérisant les liens entre un noeud et ses différents successeurs ;
- les attributs définissant les liens entre unités syntaxiques et unités lexicales.

Par exemple, pour le morceau de programme Pascal suivant:

```
procedure p (var x,y: integer);
    begin
        if x>y then x:=x-y
        else p(y,x) ;
    end;
```

on aura la représentation interne suivante:

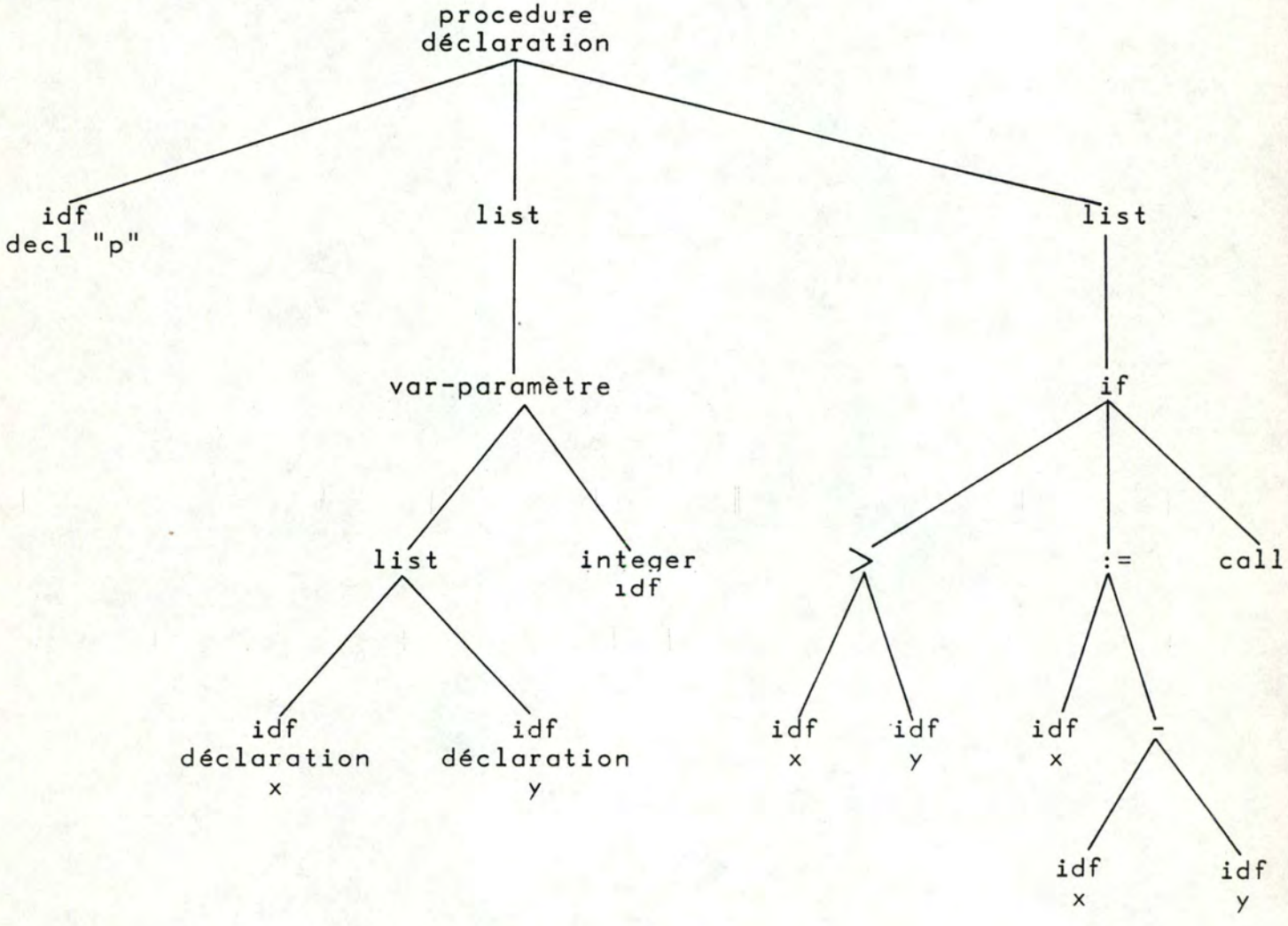


fig.1.5 Arbre syntaxique : exemple.

Il existe des liens auxiliaires qui ont été ajoutés pour des questions d'efficacité. Par exemple, il existe un lien entre une déclaration d'une variable et toutes les occurences de celle-ci. Ceci permet donc de chercher et de modifier rapidement toutes les occurences d'une variable.
La RI permet de représenter des programmes définis partiellement ou erronés. A cet effet, un noeud vide peut remplacer tout autre noeud.
Un attribut d'erreur est associé à chaque noeud pour garder une trace des erreurs sémantiques. Cet attribut peut indiquer par exemple qu'une variable n'est pas déclarée.
Pour un complément d'information sur la RI, le lecteur peut consulter (Rou, 84).

L'éditeur d'ADELE.

L'éditeur d'ADELE permet d'introduire ou de modifier des programmes interactivement. Il est basé sur les mêmes principes que le "Program Synthetizer" de Cornell (Tei, 81). L'éditeur connaît la grammaire du langage source. Ses commandes (insertion, destruction, modification) portent sur les unités syntaxiques de ce langage.
Par exemple, après avoir tapé ".IF", le schéma

```
if C
   then I1
   else I2
```

apparaît à l'écran. Puis, en utilisant certaines commandes de désignation, l'usager peut sélectionner des parties de ce schéma et recommencer le processus. De cette façon, il est impossible de produire des textes syntaxiquement incorrects.
Quand il est difficile ou gênant de procéder de cette façon, pour des opérations arithmétiques par exemple, l'utilisateur peut entrer une partie du texte; cette partie est analysée

dans son contexte, immédiatement après la frappe, et les erreurs peuvent être corrigées.

L'interpréteur.

L'exécution et la mise au point de modules individuels sont réalisées sous le contrôle d'un interpréteur de la RI. Le texte interprété peut être écrit partiellement ou de façon erronée. L'interprétation s'arrêtera sur

- une instruction de mise au point ;
- un noeud incomplet ;
- un noeud incorrect ;

A chaque pose, l'usager peut éditer ou modifier les valeurs courantes des variables.
Le debugger est de type classique. L'usage en est rehaussé par l'utilisation des facilités graphiques qui permettent d'avoir quatre fenêtres différentes sur un écran: une est destinée au texte du programme, une autre à la structure et la trace de variables, une troisième aux entrées/sorties du programme interprété et enfin une dernière destinée aux commandes de l'interpréteur.
Nous renvoyons à (Len, 84) le lecteur désireux d'approfondir le sujet.

1.2.2. Le générateur de code.

Les postes de travail utilisés dans Adèle ne sont pas nécessairement uniformes, surtout en ce qui concerne le processeur central. En effet, on compte utiliser de SM90 (Fin, 82), des Micromega (Tho, 82) et des Buroviseurs (Naf, 79); bien entendu, ce choix peut évoluer pour inclure d'autres machines.
Le générateur de code multi-machine est destiné à compiler un seul langage source pour un grand nombre de machines

cibles.

Le problème de la génération de code est l'un des problèmes majeurs en compilation: cette difficulté est accentuée dans ce cas-ci par la diversité des machines cibles. Pour cette raison, il a été décidé de procéder en deux étapes:

a) développement d'une méthode systématique de construction relativement rapide de générateurs de code. Cette méthode doit prendre en entrée une description de la machine cible, l'analyser et l'organiser de façon à construire des tables qui doivent piloter le générateur de code. Ces tables doivent contenir toutes les informations nécessaires à la traduction des opérateurs de la RI Adèle en séquences de code de la machine cible.

b) réalisation d'un système de construction automatique de générateurs de code, basé sur la méthode développée dans la première étape.

La figure ci-dessous présente le schéma général d'un tel système de génération de code:

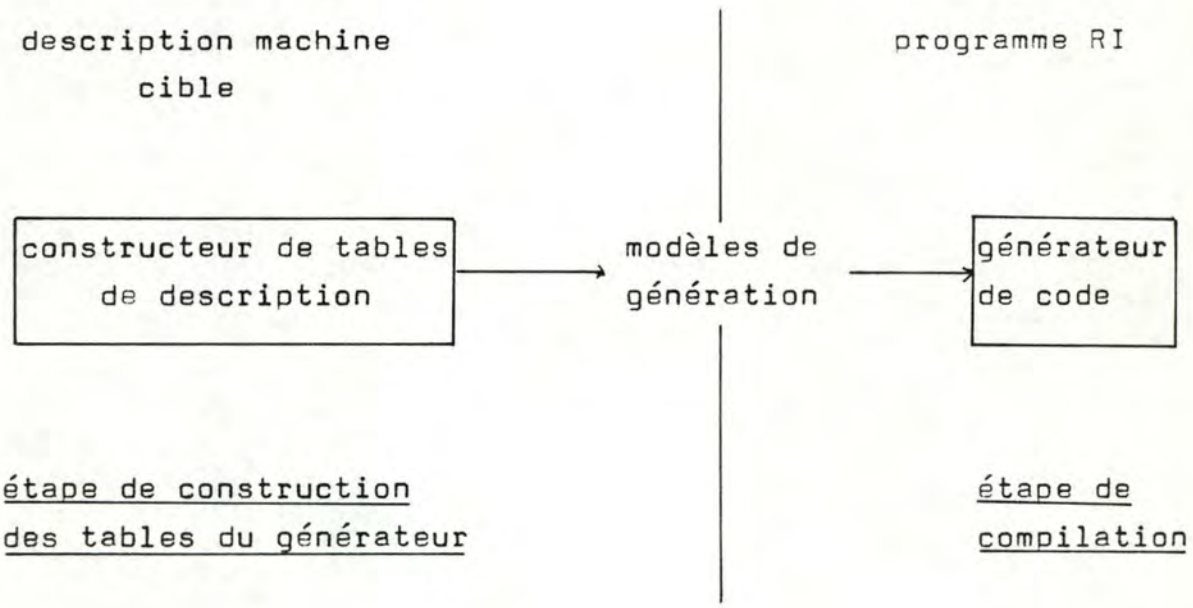


fig.1.6 Système de génération de code.

L'objectif primordial du système de génération est de produire du code de bonne qualité, au minimum comparable à celle des compilateurs disponibles sur le marché et si possible meilleure. Pour atteindre cet objectif, l'effort a été concentré sur deux aspects:

a) la sélection d'instruction machine pour une opération donnée. Le but est d'exploiter au mieux les modes d'adressage et les instructions spécialisées de la machine cible: ainsi, le code produit sera localement "optimal".
b) les optimisations sur le code généré. Le but est d'analyser les relations entre les séquences d'instructions voisines et de les exploiter pour améliorer le code. Pour s'en convaincre, regardons l'exemple suivant:

soit: les instructions

```
i := j + k ;
j := i - 5 ;
```

un compilateur simple pour cette séquence et pour un PDP-11, les instructions suivantes:

```
(1) MOV k,R0
(2) ADD j,R0
(3) MOV R0,i
(4) MOV i,R0
(5) SUB #5,R0
(6) MOV R0,j
```

Nous pouvons remarquer que la quatrième instruction est redondante puisque R0 contient déjà la valeur i. Elle peut donc être supprimée.

Pour plus de renseignements, nous renvoyons le lecteur à (San, 84).

1.2.3. Interface-usager.

Les outils de l'atelier dialoguent avec l'utilisateur par le biais du compositeur et du médiateur: Le compositeur gère une structure arborescente d'affichage, l'arbre de boîtes, tandis que le médiateur prend en charge les commandes de l'utilisateur, et détermine l'outil concerné (les outils de l'atelier sont tous potentiellement actifs à tout instant).

Les interactions transitent par deux autres composants, qui définissent les notions d'appareil logique et d'appareil virtuel.

A chaque application de l'atelier correspond un et un seul appareil virtuel, la libérant de tout problème de compétition d'accès au terminal logique. L'appareil virtuel utilise la notion classique de fenêtre, définie comme une portion de l'appareil logique. A tout instant, une application possède un certain nombre de fenêtres, visibles ou invisibles, et en est l'unique propriétaire. La donnée d'une fenêtre est donc suffisante pour déterminer le destinataire d'une commande.

La notion d'appareil logique permet de masquer les limitations des terminaux physiques. L'appareil logique de l'atelier est composé actuellement de:

- un écran alphanumérique possédant quelques possibilités semi-graphiques: surbrillance, clignotement...

- un organe de désignation: curseur ou souris ;

- un clavier composé de trois parties distinctes: touches alphanumériques, touches de désignation d'un objet et touches de désignation d'une commande.

Cette partition permet une distinction nette entre texte, objet et opérateur. En pratique, la distinction peut s'effectuer en utilisant des touches spéciales du terminal physique: "escape", "control",...

Le schéma de l'interface-usager est donc le suivant:

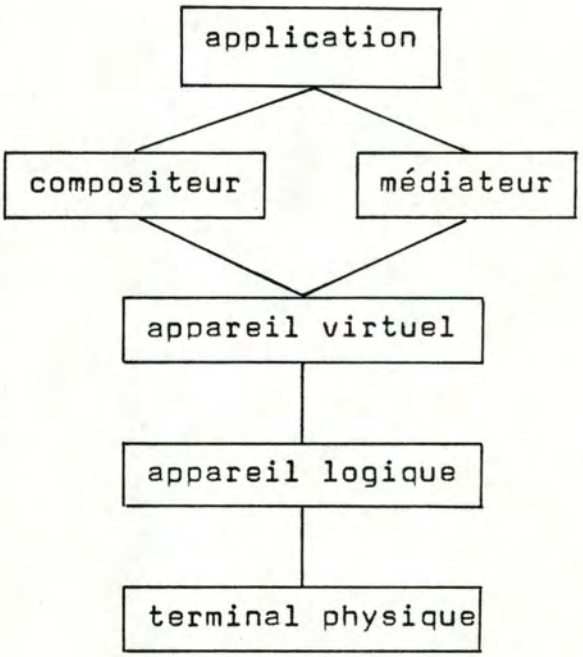


fig.1.7 Interface-usager.

1.2.3.1. Compositeur.

Le compositeur a pour but de décharger les applications de tout problème de visualisation des objets qu'elles manipulent, en fournissant la notion de boîte. En entrée, une boîte définit une unité d'information désignable à l'aide des dispositifs de sélection. En sortie, elle exprime des relations entre les informations à afficher, et permet une spécification dynamique d'attributs visuels.

La structure arborescente permet un mécanisme d'héritage des attributs: les attributs d'une boîte sont propagés à toutes ses descendantes qui ne les possèdent pas.

Les attributs de composition définissent l'organisation des boîtes dans une fenêtre:

- la composition horizontale H impose un alignement horizontal des boîtes ;

- la composition verticale V impose un alignement vertical ;

- H ou V indique une composition horizontale si possible, ou sinon verticale ;

- H et V indique un remplissage des lignes horizontales autant que possible, avec passage à la ligne, si nécessaire: mais une boîte fille ne doit pas se trouver à cheval sur deux lignes.

Les attributs visuels permettent de spécifier la fonte et la couleur des caractères et du fond, ainsi que les effets spéciaux: clignotement, surbrillance,... Enfin, l'attribut de corrélation permet à l'application de retrouver à quel objet correspond un sous-arbre de boîtes.

L'exemple suivant montre l'arbre de boîtes correspondant à un programme Pascal, et présente deux affichages distincts, suivant la taille de la fenêtre support:

```
while  x > y  do  x  :=  x - y
```

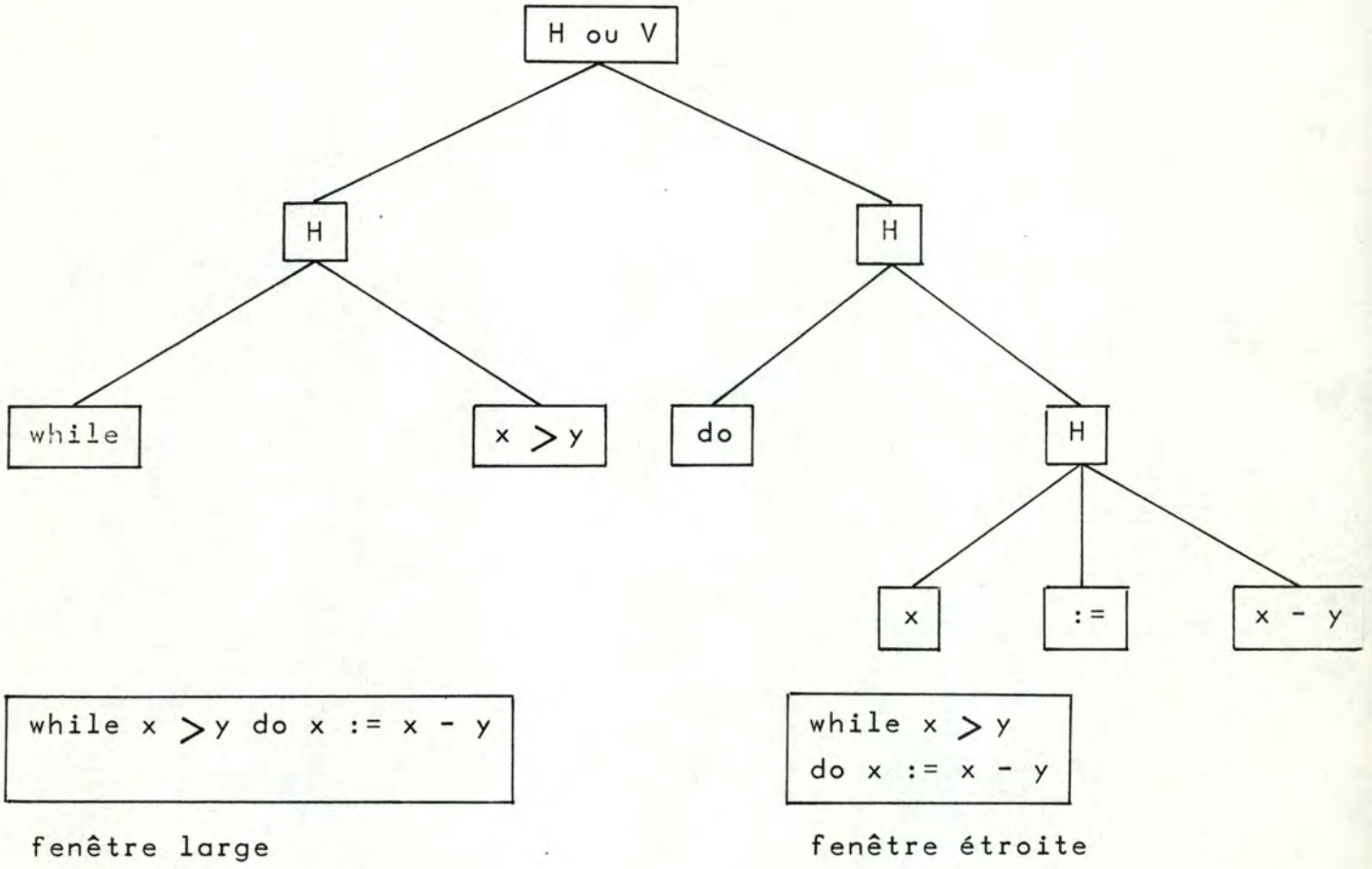


fig.1.8 Arbre de boîtes: exemple.

Le concepteur d'une application doit réaliser un compromis entre la "taille" d'une boîte feuille et celle de l'arbre des boîtes. Ici, tout caractère désigné dans le texte "x-y" représente le même objet de l'application: il n'est pas possible de désigner plus finement une partie de l'expression. Cela aurait été possible en définissant l'arbre comme une composition horizontale des boîtes feuilles "x", "-" et "y". Mais l'arbre complet du programme serait alors beaucoup plus important.

L'affichage est incrémental: dès que l'application modifie une partie de l'arbre, le compositeur calcule la nouvelle image à visualiser, en évitant bien sûr d'effacer l'écran et de tout réafficher.

1.2.3.2. Médiateur.

Le médiateur définit un modèle d'interaction entre l'application et l'usager, dont les grands principes sont:

- Emploi d'opérateurs universels: par exemple, l'action de destruction d'un objet porte toujours le nom "détruire", quelle que soit l'application concernée. Le nombre de ces opérateurs est volontairement limité pour en faciliter l'apprentissage. Par exemple, l'action d'imprimer un programme consiste à le copier dans l'objet imprimante.

- Redondance de la désignation: un objet, comme un opérateur, peut être désigné de trois manières: sélection d'une représentation visuelle, frappe de son nom, ou emploi d'une touche "synonyme" accélérant le processus de la frappe. L'utilisateur peut ainsi choisir à tout moment le mode de désignation qui lui convient le mieux.

- Gestion des fenêtres: le médiateur contrôle l'organisation de l'écran logique. Les applications lui demandent la création et la destruction d'une fenêtre et il en détermine la localisation et la taille initiale suivant la taille de l'écran. De plus, l'utilisateur peut à tout moment modifier les choix du médiateur.

- Possibilité de modifier un objet ou un opérateur: le médiateur conserve une liste des propriétés associées à chaque objet et opérateur. Pour un opérateur, cette liste indique le mode d'interaction: novice ou expert, avec ou sans confirmation, avec ou sans création de fenêtres,.. A un type d'objet correspond la liste des opérateurs qui lui sont applicables.

L'utilisateur peut ainsi décider à tout moment que la destruction d'un objet "programme" nécessite une confirmation, alors que celle d'un objet "instruction" n'en nécessite pas. De même, le mode novice entraîne l'utilisation d'un formulaire guidant l'usager pas à pas, tandis que le mode expert limite le dialogue au strict minimum.

Pour tout complément d'information sur l'interface-usager, le lecteur peut consulter (Her, 84).

1.2.4. La base de programmes.

Tous les objets gérés par Adèle (programmes, documentation ...) sont conservés dans une base de données. Dans cette base sont représentées les différentes relations exprimant la composition de systèmes à partir de modules, et la cohérence des systèmes à versions multiples.
C'est dans ce cadre que nous avons réalisé notre travail. Nous développerons donc davantage cette partie à la section 1.3..

1.3. La base de programmes. (Est, 84), (Gho, 84).

1.3.1. Objectifs.

Dans un contexte de développement modulaire de gros logiciels, des centaines, voire des milliers de modules peuvent coexister. Ceux-ci n'entrent pas seulement en relation entre eux mais aussi avec des objets différents tels que leur spécification, leur documentation, leur code binaire...
C'est dans ce cadre d'entités et d'associations que l'emploi d'une base de programmes devient nécessaire. Cette nécessité s'avère d'autant plus importante que chaque module peut exister en versions multiples, et chaque version peut comprendre plusieurs révisions.

Dans le cadre d'ADELE, une telle base a dès lors été mise au point. Sa réalisation a surtout été guidée par les objectifs suivants:

- Constitution de façon sûre et efficace de configurations complexes du logiciel. On verra comment l'utilisateur peut, au moyen d'un langage simple, exprimer les caractéristiques souhaitées de sa configuration. L'expression de ces caractéristiques permettra au système de gestion de la base de retrouver tous les modules concourant à la constitution de cette configuration.

- Maintien de la cohérence des différents objets. Toute action menée sur la base doit pouvoir être analysée de manière à prendre les mesures nécessaires pour éviter les effets de bord éventuels.
Nous ne pouvons à ce stade donner trop d'explications à ce sujet étant donné que nous ne connaissons pas encore les différents types d'objets maintenus dans la base. En ce qui concerne une configuration d'une famille de systèmes, elle sera dite cohérente si ses différents composants ne présentent pas d'incompatibilités.

Considérons le cas suivant:

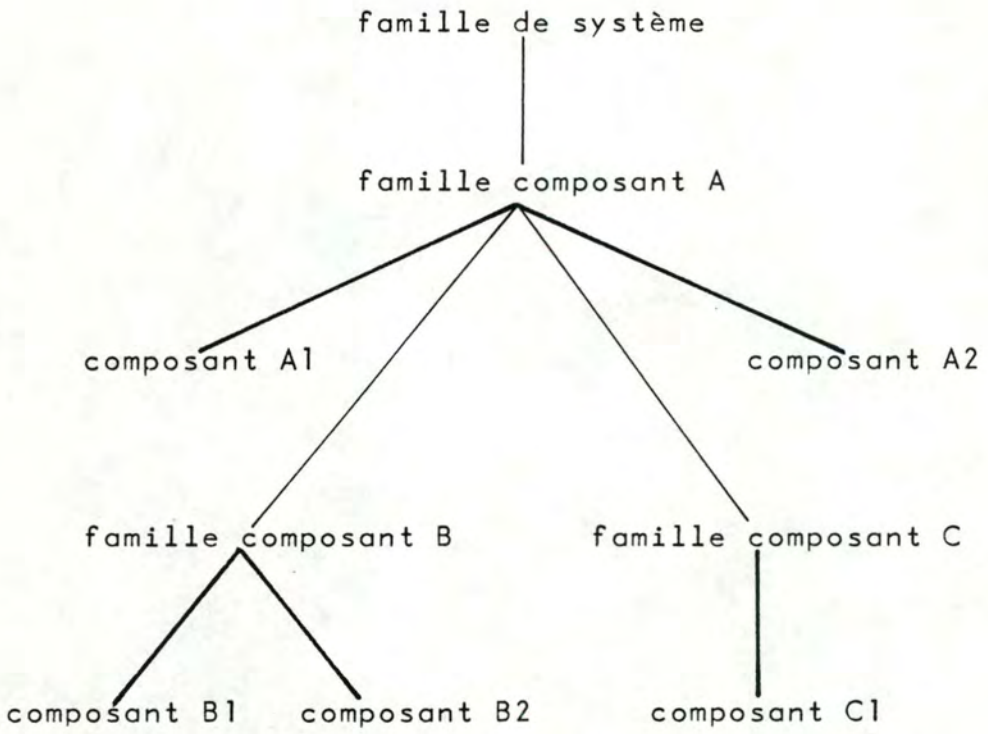


fig. 1.9 Famille de systèmes. ——: utilise
——: comprend

Il est évident que l'agencement de différents composants pour former une configuration n'est pas arbitraire. Si les composants A1 et B2 sont incompatibles, toute configuration les incluant sera incohérente.
Nous verrons dans un paragraphe spécialement dédié à cette notion, comment la cohérence se généralise aux différents objets de la base.

- Protection de chaque partie du logiciel en cours de développement. Tout programmeur ne pourra accéder qu'à ce qui le concerne directement.

1.3.2. Objets et relations dans la base.

1.3.2.1. Interfaces, corps et relations.

Une des conséquences de la modularité résulte en la décomposition du système en composants distincts communément appelés modules. Dans ADELE, tout module est entièrement défini par sa spécification fonctionnelle. D'un point de vue pragmatique, il est représenté par l'association d'un corps et d'une interface. La notion d'interface est très semblable à celle de "définition" dans MODULA2 (Wir, 82) ou de "module" dans MESA (Mic, 79). Elle représente la partie visible d'un module.
Si on se réfère à (Par, 77), une interface entre deux composants logiciels doit inclure toutes les hypothèses que chacun d'eux fait à propos de l'autre.
Dans ADELE, elle reprend les différentes ressources que le module englobant met à la disposition des modules extérieurs. Une ressource peut se définir comme un moyen dont on peut disposer pour conduire une action. Une ressource mise en oeuvre par un module A sera dès lors considérée comme un moyen pour un module B de remplir sa fonction si elle est décrite dans l'interface entre A et B.
Il est clair qu'une description standard de ces ressources et indépendante de tout langage de programmation serait une description idéale. Les interfaces pourraient ainsi être pratiquement couplées à la phase de spécification et totalement indépendantes de la programmation. On pourrait définir un Méta-Langage de description et écrire un pré-processeur qui se chargerait de la transcription dans le langage utilisé.
Dans ADELE, bien que la notion d'interface ne soit pas encore réellement mise au point, on tend vers la solution inverse, c'est-à-dire le strict couplage de l'interface au langage de programmation.

Dans l'exemple d'une gestion de pile d'entiers naturels, nous aurons en Pascal l'interface suivante:

```
procedure pop (p:pile);
procedure push (p:pile, n:integer);
procedure top (p:pile; var n:integer);
```

ou encore

```
procedure pop (p:array [1..20] of integer);
procedure push (p:array [1..20] of integer, n:integer);
procedure top (p:array [1..20] of integer;
                    var n:integer);
```

où - la procédure pop supprime l'élément du sommet de la pile s'il existe,
- la procédure push dépose un élément au sommet de la pile si elle n'est pas pleine,
- et la procédure top renvoie l'élément au sommet de la pile s'il existe.

La différence entre ces deux interfaces est évidente. La première présente aux modules extérieurs un type "pile" et la seconde révèle que la pile est implémentée par un tableau limité à 20 éléments. Ce deuxième exemple est dans une certaine mesure dommageable. Les modules extérieurs connaissent l'implémentation de la pile alors que cette information ne leur est pas pertinente. D'un point de vue pragmatique, cela signifie qu'ils peuvent utiliser ces informations, et il en résulte que tout changement d'implémentation s'accompagnera éventuellement de modifications de modules extérieurs.
La première interface est en fait plus générale. Elle correspond à ce que D.L. Parnas appelle une "interface abstraite". Elle représente en effet plusieurs interfaces possibles. La seconde description en est un exemple.
Dans la première interface, le type "pile" est abstrait et cela

présente l'avantage qu'il peut être implémenté de diverses manières sans qu'aucun des modules extérieurs ne soit impliqué.
Si nous avons restreint nos exemples à un ensemble de définitions de procédures, il ne faut pas en déduire qu'une interface est réduite à ces seules définitions. Dans le cadre de Pascal, elles peuvent aussi comprendre des constantes, des types, des variables et des fonctions. Notons également qu'une interface est associée à une spécification fonctionnelle et qu'elle est facilement extensible. Si un concepteur décide que le calcul de la profondeur d'une pile peut être utile, il peut ajouter dans l'interface la définition suivante:

```
fonction depth (p:pile): integer;
```

Remarquons finalement que l'interface ne reprend que les objets globaux et exclut de son contenu tout objet local au module qui la comprend.
Nous avons déjà mis l'accent sur la relation existant entre le langage et l'interface, et nous utiliserons dorénavant Pascal pour illustrer nos explications.
Toutes les ressources fournies dans l'interface sont implémentées ou réalisées par le corps associé. Cette relation de réalisation du corps vers l'interface peut s'exprimer d'une façon générale: réaliser une ressource consiste à lui affecter une valeur. Cette valeur dépend du type de ressource à réaliser et si nous étudions cet aspect plus localement à travers chaque type de ressource, nous pouvons dire que:

- on réalise une constante en lui conférant une valeur numérique, alphanumérique, ou alphabétique;

- on réalise une variable en lui affectant une valeur conforme à son type;

- on réalise un type en lui attribuant la valeur

qui le définit. La réalisation du type "pile" peut se définir en écrivant:

```
type pile = packed array [1..20] of integer;
```

- on réalise une procédure ou une fonction en l'implémentant, c'est-à-dire en transcrivant l'algorithme, qui réalise ses spécifications fonctionnelles, dans un langage de programmation.

Un corps contiendra dès lors la réalisation explicite de ses constantes, types, procédures et fonctions, les variables étant réalisées dynamiquement lors de l'exécution.
Si nous possédons l'interface suivante

```
const hauteur = 20;
type pile = array [1..hauteur] of integer;
var p:pile;
procedure pop (p:pile);
procedure push (p:pile, n:integer):
```

un corps la réalisant pourrait être de la forme:

```
Program Gestion-Pile;

const hauteur = 20;
type pile = array [1..hauteur] of integer;
var nb-element : 1..hauteur;
    p:pile;
procedure pop (p:pile);
begin
    if nb-element > 0 then
         nb-element := nb-element - 1;
end;
procedure push (p:pile, n:integer);
begin
```

```
        if nb-element < hauteur then
        begin
            nb-element := nb-element + 1;
            p [nb-element] := n
        end;
    end;

    begin
    end.
```

Deux remarques essentielles sont à faire à propos de ce corps. Premièrement, la dernière paire "begin-end" n'est nécessaire que pour des raisons de compilation.
Deuxièmement, le paramètre "p" des procédures ne semble pas utile. La pile est en effet entièrement définie dans ce corps et un passage de paramètre par valeur tel qu'il est représenté ici peut conduire à des erreurs. Un passage par nom, bien qu'il n'existe pas en Pascal, serait plus adéquat: il serait cependant impératif d'appeler les procédures par: pop ($\underline{\underline{p}}$) ou push ($\underline{\underline{p}}$,n).
Si nous avons gardé ce paramètre, c'est pour une raison fondamentale. L'utilisateur de ces procédures push et pop désire ajouter ou supprimer un élément au sommet d'une pile. Tout ce qu'il connaît est concentré dans l'interface et sa spécification et si "p" n'est pas utilisé dans "pop", l'utilisateur n'en sait rien. La réalisation de la procédure lui est cachée. Le réalisateur aurait par exemple pu décider de remettre l'élément supprimé à Ø avant de décrémenter nb-element d'une unité.
D'autre part, pour réaliser ses propres ressources, un corps peut soit se suffire à lui-même, soit utiliser les ressources d'une interface réalisées par un corps extérieur. On dit dans ce dernier cas que le corps utilise ces ressources extérieures, ou encore qu'il dépend d'elles.
Nous dirons qu'un composant A utilise ou dépend d'un composant B si le bon fonctionnement de A nécessite la présence et le bon fonctionnement de B. Dans le cadre d'ADELE, nous dirons

donc qu'un corps A utilise ou dépend d'une interface B si le bon fonctionnement de A nécessite la présence et le bon fonctionnement du corps qui réalise B.
Tout comme l'interface est associée à une spécification fonctionnelle, le corps est associé à une spécification de réalisation. Soulignons également qu'un utilisateur ne peut distinguer un simple corps d'un système complexe si ces deux éléments lui présentent la même interface. Toute modification d'un corps est par ailleurs transparente si elle n'induit aucune transformation de l'interface.

Nous avons jusqu'à présent défini 4 types d'objets et 4 types de relations que nous pouvons résumer par le schéma E/A suivant:

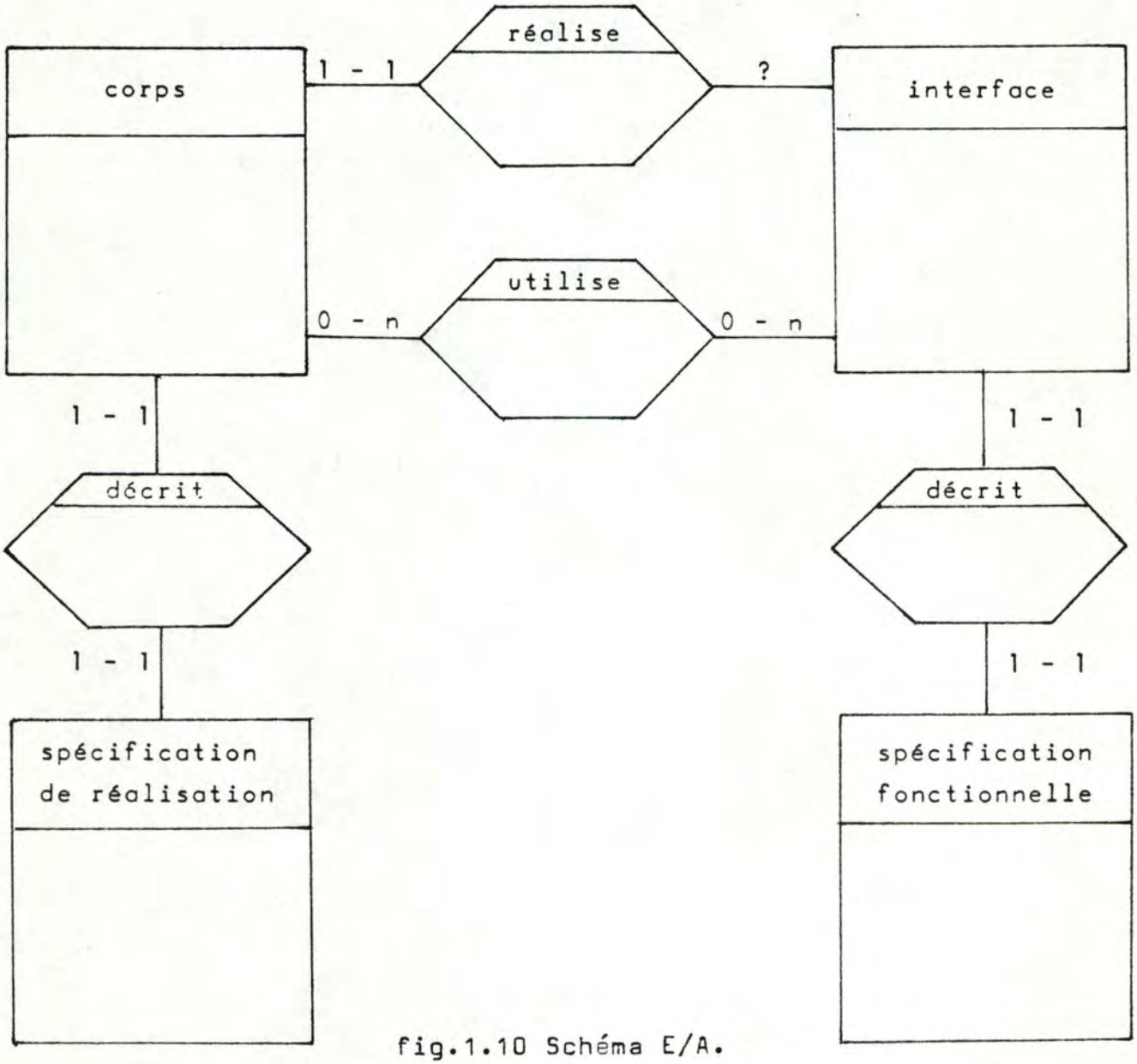
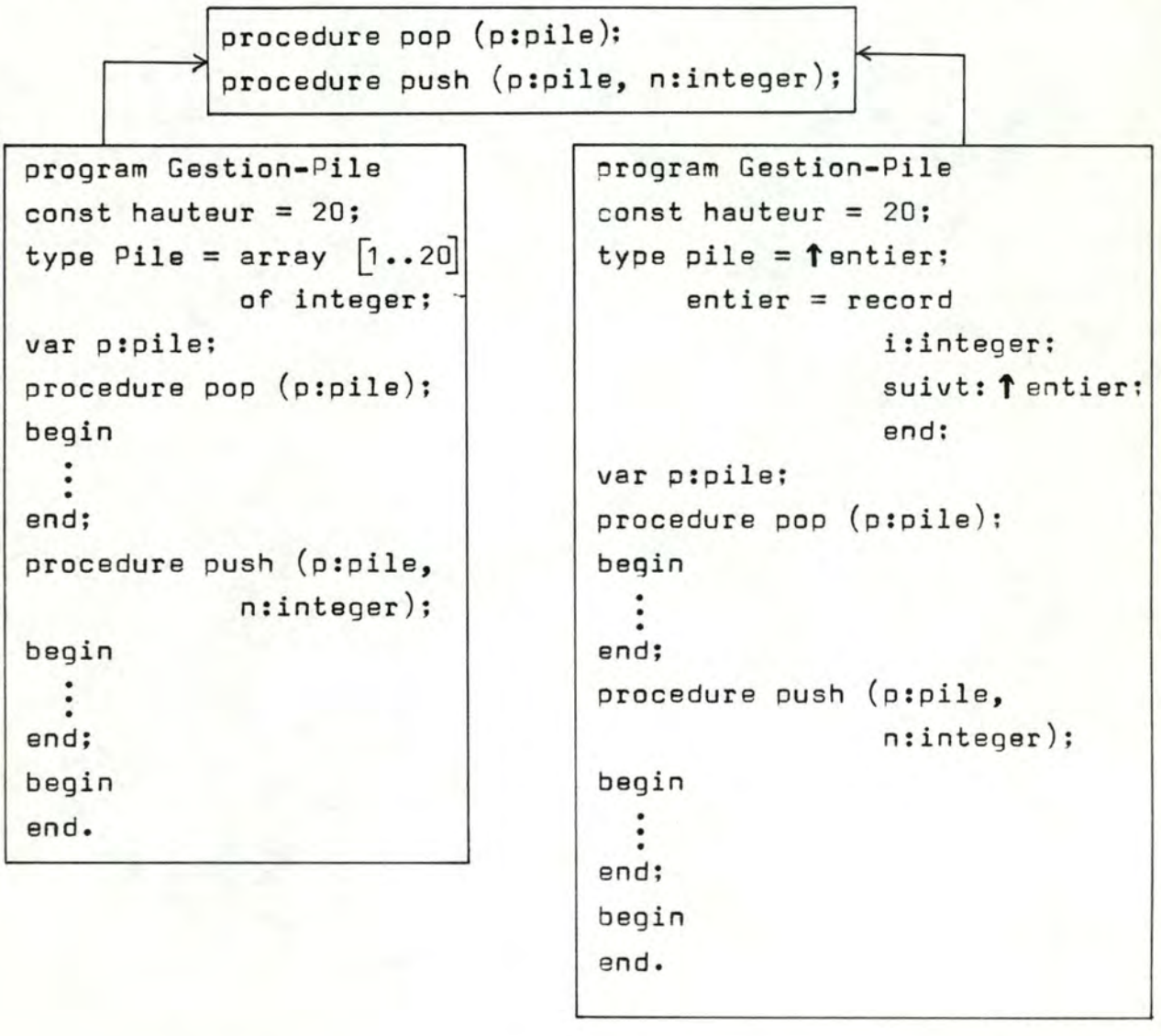


fig.1.10 Schéma E/A.

Le point d'interrogation de la connectivité d'interface dans l'association "réalise" n'est pas révélé parce qu'elle fait l'objet du paragraphe suivant.

1.3.2.2. Versions et révisions.

La connectivité de l'interface dans la relation "réalise" qui l'associe au corps est en réalité 1 - n. Cela signifie qu'une interface est réalisée par au moins un corps. Chacun des corps réalisant est appelé une version de réalisation de l'interface. Dans le cadre de la gestion d'une pile, ce principe peut être illustré de la façon suivante:



⟶ : réalise

Ces deux versions se caractérisent par une différence dans la structure de données. Dans l'une, la pile est un tableau d'entiers, dans l'autre, c'est une liste de pointeurs. On peut facilement trouver d'autres exemples où la structure de données reste stable, et où les algorithmes sont modifiés. Un tableau d'entiers peut, par exemple, être trié de multiples façons: on connaît les tris par insertion, sélection, échange...
A chaque méthode correspond une version de réalisation différente.
D'autre part, lors du développement ou de l'utilisation de systèmes, il est fréquent que de nouveaux besoins surgissent de l'expérience. Leur prise en compte conduit à la réalisation d'une nouvelle version du système et ne nécessite souvent la modification que d'un nombre restreint de modules. Pour que les différentes versions du système puissent coexister, il suffit de créer une nouvelle version des modules impliqués plutôt que de les remplacer. On aboutit ainsi à un système existant en une version de base, et se multipliant en versions "sur mesure". C'est dans cet éventail de versions ou cette famille de systèmes, qu'un client pourra choisir celui qui correspond le mieux à ses besoins.
Comme dans RCS (Tic, 82) ou SCCS (Roc, 75), la base ADELE offre un second degré de variabilité: la révision.
Les révisions successives d'une version en reflètent l'évolution chronologique. Selon (Kra, 82), "la distinction entre version et révision reste encore arbitraire. En principe, les versions correspondent à des modifications ayant une signification logique (par exemple, un changement de configuration, ou le passage d'une version de mise au point à une version de production), tandis que les révisions correspondent à des modifications plus mineures, résultant d'améliorations ou de corrections d'erreurs".

A notre avis, une nouvelle version peut résulter de différentes situations:

- le changement radical d'une structure de données;

- la modification de l'algorithme (tri par insertion ⟶tri par échange) qui risque de modifier l'efficacité ou la complexité;

- le besoin d'inclure la nouvelle version dans une nouvelle configuration. En effet, deux révisions d'une version ne peuvent appartenir à des configurations différentes alors que rien ne l'empêche pour deux versions distinctes même si elles réalisent la même interface.

Il apparaît donc que les différences entre versions sont d'un ordre plus sémantique alors que les différences entre révisions successives privilégient plutôt la syntaxe, et altèrent dans une moindre mesure la sémantique par des améliorations locales (remplacer un while par un repeat, par exemple).

Nous maintiendrons, en conséquence, des schémas du type suivant dans la base:

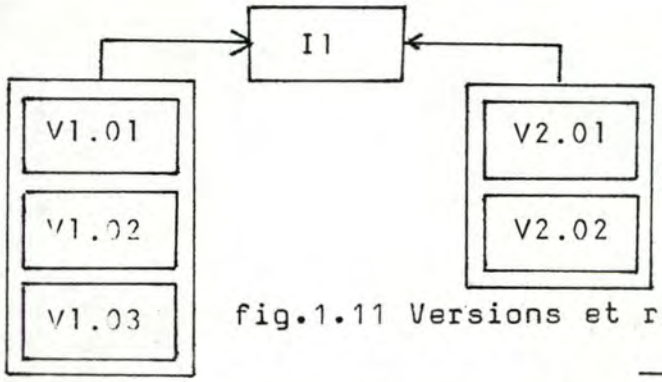


fig.1.11 Versions et révisions

⟶: réalise

Les versions V1 et V2 contiennent chacune plusieurs révisions et réalisent toutes deux l'interface I1.

Outre cette notion de version de réalisation, ADELE introduit le concept de version d'interface. L'une et l'autre nous conduisent à définir un nouveau type: la famille.

L'utilisation de ce principe peut favoriser la réduction des versions d'interface. Gérer une multitude d'interfaces peut s'avérer dommageable, ne fut-ce que par le fait que la somme de connaissance à propos du logiciel s'en trouve accrue.
Le principe peut être résumé comme suit:

- produire une interface générale qui reste valable pour plusieurs interfaces possibles;

- cette interface révèle les caractéristiques communes aux diverses interfaces possibles, et en cache les différences;

- elle est, à ces titres, dite "abstraite".

Ce principe peut être illustré par l'exemple suivant:

- soit un système d'exploitation à utiliser dans une application quelconque.

- On désirerait que l'application reste, dans la mesure du possible, indépendante de ce système. On envisage par exemple de la transférer vers des systèmes différents.

- Il suffit de: - créer une interface indépendante du système et qui reprend des fonctions "souhaitées";
- écrire le corps qui réalise cette interface en fonction du système utilisé.

Ce qui se résume de la façon suivante:

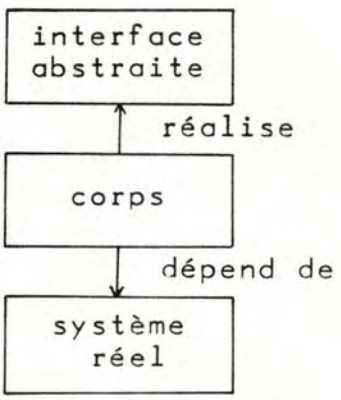


fig.1.12 Interface abstraite.

1.3.2.3. Familles.

Lors du développement d'un logiciel, il est fréquent que des modifications d'interface soient nécessaires. Elles ont pour conséquence le remplacement de l'interface existante ou la création d'une nouvelle version de cette interface.
Le premier cas peut être illustré par l'ajout de la fonction depth à l'interface proposée au paragraphe précédent et le second cas peut faire l'objet de l'illustration suivante:

supposons que le réalisateur ait produit la seconde interface présentée au paragraphe précédent au sujet de la gestion de pile. Cette interface révélait que la pile était implémentée par un tableau de 20 éléments. Si on décide de transformer cette structure de données en une liste de pointeurs tout en conservant l'implémentation précédente, une nouvelle interface sera nécessaire. Nous aurons dès lors le schéma suivant:

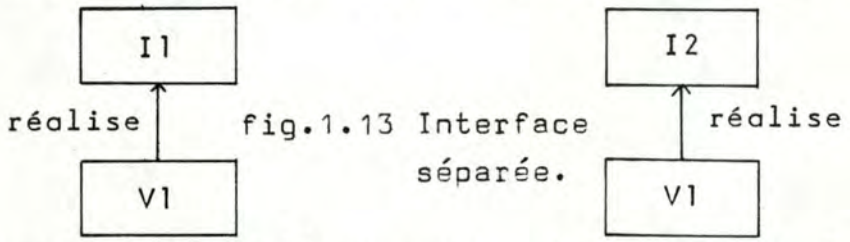


fig.1.13 Interface séparée.

au lieu du schéma présenté au paragraphe précédent, à savoir

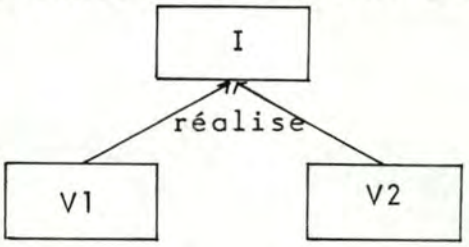


fig.1.14 Interface commune.

D'autre part, le transfert d'un logiciel sur des machines, des systèmes ou des applications différents induit souvent des modifications d'interfaces.
Le Docteur D.L. Parnas a largement développé le principe des interfaces abstraites. On peut consulter à ce sujet (Par, 77), (Par, 81) et (Par, 81a).

L'interface abstraite représente en fait un système abstrait. Le système réel n'est connu que du corps et les modules extérieurs en sont, de cette façon, rendus indépendants.
Tout changement de système réel nécessitera la seule modification du corps qui réalise en fait le "mapping" entre l'abstraction et la réalité.
Ce principe n'est cependant pas le remède miracle. Il peut arriver que de nouvelles interfaces soient nécessaires. Des exemples en sont donnés dans (Par, 81).

L'exemple de la gestion de pile introduisait l'alternative suivante:

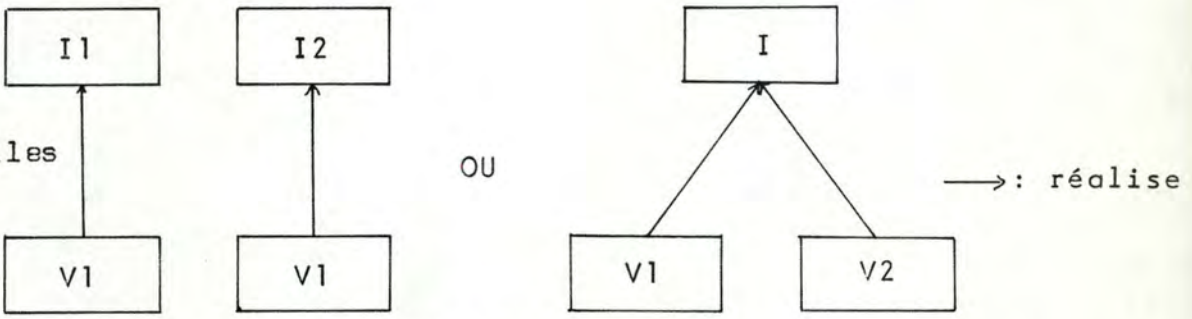


fig.1.15 Familles

Quelle que soit la solution choisie, nous dirons que les modules (I1,V1) et (I2,V2), ou (I,V1) et (I,V2) appartiennent à la même famille. On appelle "famille" l'ensemble de ces interfaces et des corps qui les réalisent.
Bien que cela puisse paraître paradoxal vis-à-vis de la définition de module (cfr. 1.3.2.1.), une famille apparaît donc comme l'ensemble des versions possibles d'un même module.
Elle est schématisée de la façon suivante:

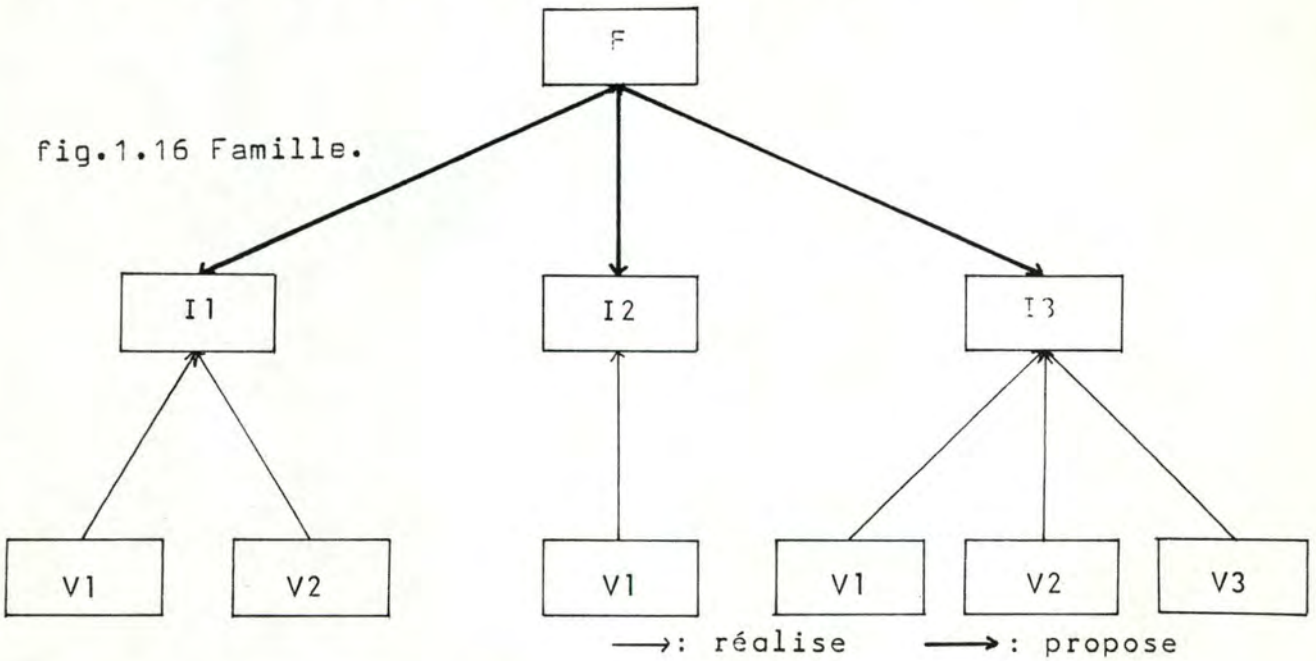


fig.1.16 Famille.

Dans cet exemple, la famille F "représente" un module offrant trois interfaces dont I1 et I3 sont réalisées par plusieurs versions.

En conséquence de ces différentes notions de versions multiples, la définition que nous avons donnée d'un module comme étant l'adjonction d'un corps et d'une interface apparaît trop restrictive. Par la suite, nous utiliserons le terme de module de façon informelle au sens de famille. Nous emploierons également indifféremment les termes de corps, réalisation ou version pour désigner l'entité qui réalise une interface.
Par abus de langage, nous dirons qu'une famille dépend d'une autre si au moins une version de la première dépend d'une interface de la seconde.
En toute généralité, nous pouvons donc considérer toute base de programmes gérée sous ADELE comme un graphe de dépendances entre familles.

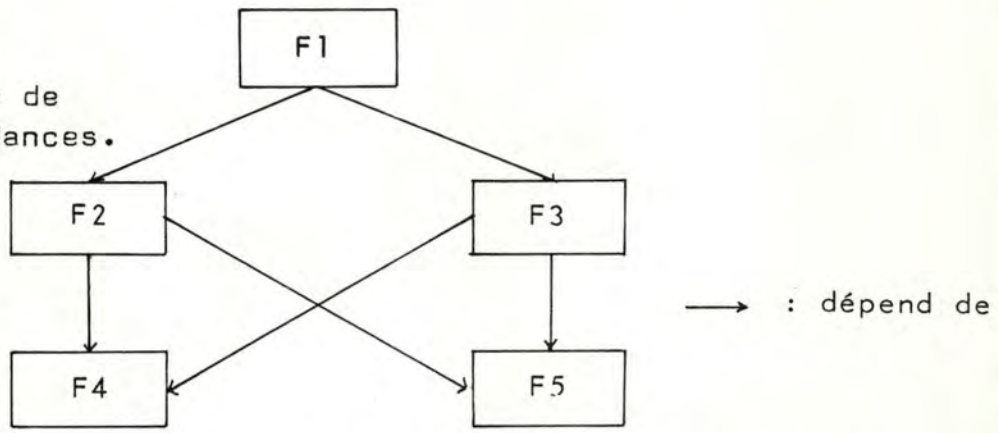


fig.1.17 Graphe de dépendances.

Tout graphe géré par la base est muni d'une racine (ici F1) et dépourvu de circuit.
Une famille est désignée par le cheminement qui permet de l'atteindre en partant de la racine du graphe. Chaque nom de familles intermédiaires intervenant dans cette désignation est précédé du signe " >". F2, par exemple, est désigné par >F1>F2. Pour F4, nous remarquerons qu'il existe deux désignations possibles: >F1> F2>F4 ou >F1 >F3 >F4.
Pour lever cette ambiguité, on introduit les concepts de

familles filles et familles externes. Si une famille a plusieurs ascendants d'ordre 1, elle sera fille d'un seul, et externe à toutes les autres. Le choix de l'ascendant père est laissé au concepteur. Par défaut, la première famille créée est fille et les suivantes sont externes, mais il n'existe aucun autre critère précis pour trancher automatiquement ce problème. On peut par exemple définir le graphe de la figure 1.17 bis:

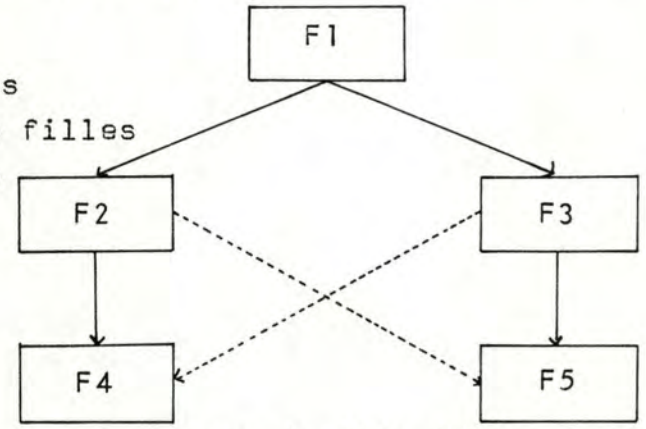


fig.1.17 bis
Familles filles
et externes

⟶: a pour famille fille ⇢: a pour famille externe

Si on ne considère ce graphe qu'à travers la relation "fille", on s'aperçoit qu'il s'agit d'une arborescence puisqu'il est simplement connexe, sans cycles(1) et muni d'une racine.
Dans la redéfinition de la figure1.17 bis, F4 est uniquement désignée par >F1>F2>F4.
Rappelons afin d'éviter toute confusion que Fi représente en fait un ensemble de versions d'un même module. F2 représente par exemple:

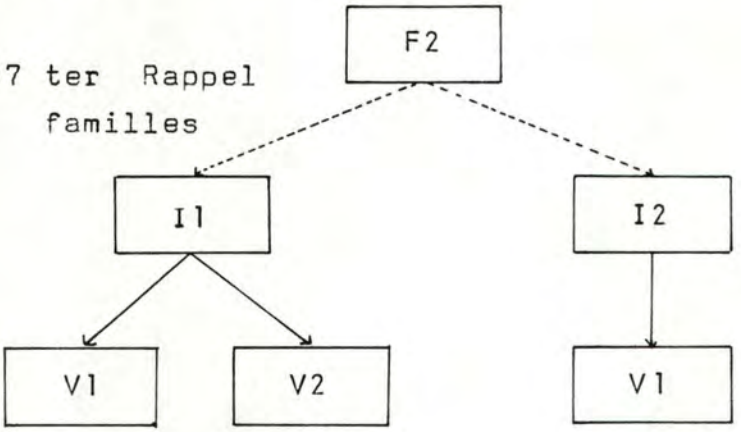


fig.1.17 ter Rappel
familles

(1) On parle de cycles pour des graphes non orientés et de circuits pour des graphes orientés. C'est pourquoi la figure1.17 bis contient des cycles mais aucun circuit.

Le fait que le graphe soit sans cycle est une condition nécessaire et suffisante pour qu'il puisse être décomposé en niveaux. Nous dirons qu'une famille du niveau i ne dépend que de familles de niveau i + n, la racine étant la famille de niveau 0. Sur la figure1.17bis, nous observons qu'on peut décomposer le graphe en trois niveaux, numérotés de 0 à 2:

niveau 0	F1	
niveau 1	F2	F3
niveau 2	F4	F5

fig.1.18 Décomposition en niveaux.

En utilisant cette nouvelle structure, nous pouvons dire qu'une famille fille appartient, dans la pratique, au niveau directement supérieur à celui contenant la famille père. Les familles externes appartiennent quant à elles à n'importe quel niveau supérieur.

1.3.2.4. Objets associés.

La gestion de ces familles, corps et interfaces impose qu'on leur associe un certain nombre d'objets. La figure1.19donne un aperçu de ces associations pour un corps. Chaque objet associé correspond à un type particulier concrétisé par une notation qui permet de l'identifier.

Objet	Objet associé	Type
corps	n révisions	text.i ($1 \leqslant i \leqslant n$)
	n codes objets	co
	1 historique	hist
	1 manuel	man
	1 documentation	doc

fig. 1.19 Objets associés.

Une documentation, un manuel, un historique et un texte sont associés à toute interface, tandis qu'à une famille ne correspondent qu'une documentation, un manuel et un historique.
Le concept de documentation n'étant pas encore clairement spécifié, nous n'en ferons pas une étude détaillée afin de ne pas dénaturer la réalité. Nous dirons simplement que l'usager doit bénéficier d'une liberté absolue et doit pouvoir définir autant de types de documents qu'il le désire. Un système de gestion automatique de documentation de programmes est en cours de mise au point.
Quant au manuel, il correspond au coeur de la base ADELE. On y concentre les informations nécessaires à la gestion des différents objets (famille, interface et corps). Il joue à la fois le rôle de descripteur et de mode d'emploi de l'objet auquel il est associé. Il se présente sous la forme suivante:

```
manuel X;
  ( <rubrique> )*
end;
```

Chaque manuel comprend plusieurs types de renseignements classés sous différentes rubriques:

- une rubrique attribut définie par l'usager précisant des propriétés de l'objet non vérifiées par le système. Ces propriétés servent à caractériser l'objet et peuvent être considérées comme un moyen complémentaire de désignation.

- une rubrique contrainte définie par l'usager précisant certaines propriétés de l'objet et certaines contraintes sur l'usage qui peut en être fait. Ces renseignements sont vérifiables et permettent à l'atelier de respecter ces contraintes et de conserver ces propriétés. Ils constituent la partie "mode d'emploi" du manuel.

- une rubrique interne contenant des renseignements sur l'objet (nature, état, localisation, ...). Ces renseigne-

ments sont fournis par l'atelier. Ils sont consultables mais non modifiables par l'usager. Ils constituent la partie "descripteur" du manuel.

Le manuel d'une famille par exemple contiendra des informations telles que le nom de ce manuel, le niveau hiérarchique de la famille, ou encore la liste des familles filles, des familles externes, et la liste de ses interfaces (relations). Ces différentes informations forment la rubrique interne, les rubriques attribut et contrainte étant absentes dans le cas d'une famille.
Dans l'exemple de la figure 1.17bis, le manuel de F2 contiendra:

```
manuel F2;
        -- informations consultables uniquement
           -- état
              nom: nom du manuel (voir 1.3.2.6. Dési-
                                  gnation des objets)
              niveau: 2
           -- relations
              liste-familles-filles: F4
              liste-familles-externes: F5
              liste-interfaces: I1 I2
end;
```

Quant aux manuels des interfaces et corps, nous ne possédons à ce stade du développement pas assez de concepts pour les expliciter d'une manière précise. Les parties intéressantes de leur contenu émergeront au cours de cette étude. Une description détaillée en sera donnée en annexe B.

Les historiques seront traités en profondeur au chapitre 2 de la deuxième partie de cet ouvrage. Disons dès à présent que l'historique d'un objet sert à relater son évolution dans le temps. Il renferme des informations telles que la date et l'auteur de création ou de modification d'un objet, le type de

l'objet créé ou modifié (text, co, man, doc), un commentaire explicitant la modification...

1.3.2.5. Espace de travail.

Pour éviter que tout programmeur ait accès à tout objet de la base, on définit la notion d'espace de travail.
Un espace de travail est un sous-ensemble de l'ensemble des familles. Ce sous-ensemble possède les propriétés suivantes:

- il est, de façon univoque, déterminé par une famille, soit Fk;

- il contient la famille Fk et toutes ses familles filles ou externes;

- il est attribué à une ou plusieurs personnes.

Les personnes affectées à un espace ne pourront accéder qu'aux familles qu'il contient. Toute personne ayant l'accès à une famille déterminée (Fk) pourra accéder à toute famille appartenant à l'espace de travail déterminé par Fk.
Il va dès lors de soi que l'administrateur ou le chef du projet recevra l'accès à la racine de la base, ce qui lui permettra d'accéder à toutes les familles sans restriction. Il pourra ensuite répartir son équipe entre les différentes couches du logiciel. La figure 1.20 est un exemple d'une telle répartition.

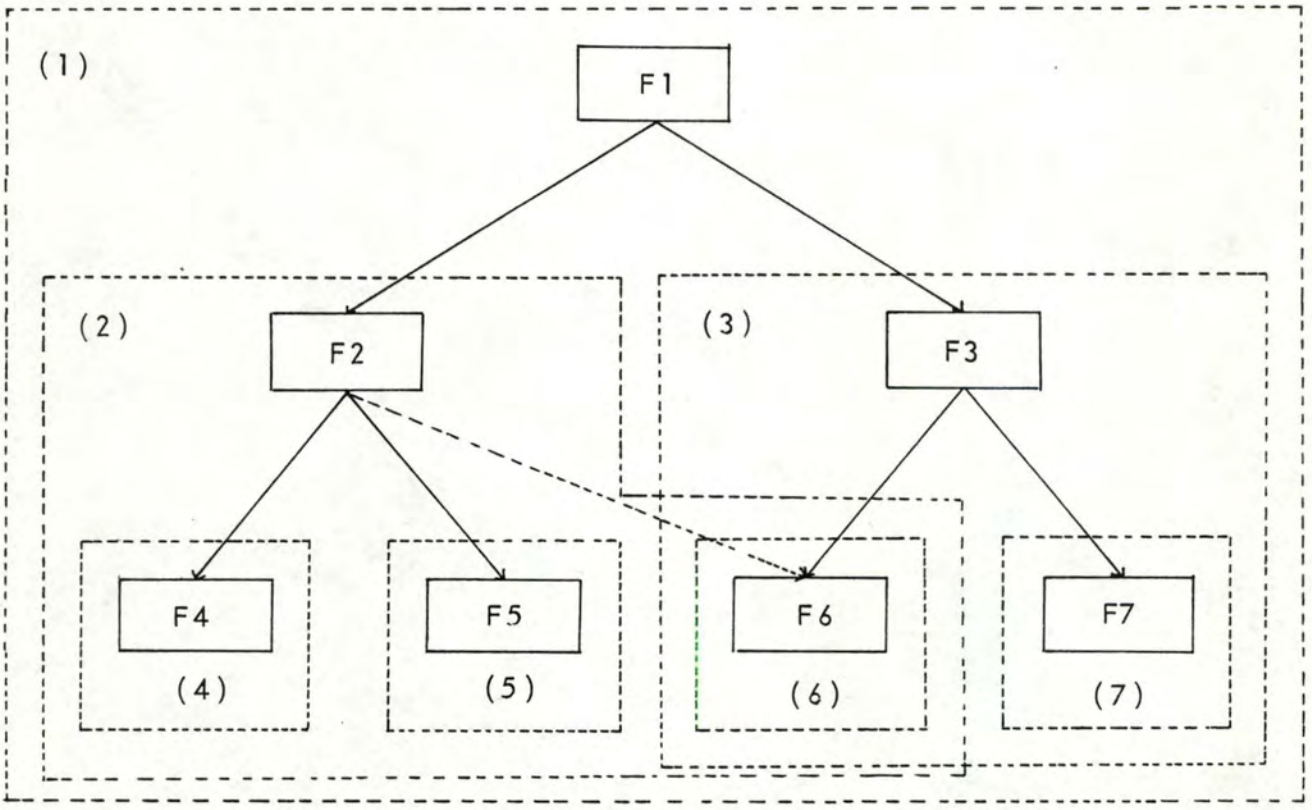


fig.1.20 Espaces de travail.

——→: a pour famille fille

------→: a pour famille externe

(1) espace affecté au chef de projet
(2) espace affecté au programmeur A
(3) espace affecté au programmeur B
(4) espace affecté au programmeur C
(5) espace affecté au programmeur D
(6) espace affecté au programmeur E
(7) espace affecté au programmeur F

En terme de la relation "utilise", on remarque que toute personne a accès à tout ce qu'elle utilise. Ainsi, il est possible que deux programmeurs différents aient accès à la même famille (F6 pour A, B et E).
Quant aux restrictions d'accès, on constate par exemple que A ne peut accéder à F1, F ne peut accéder qu'à F6,...

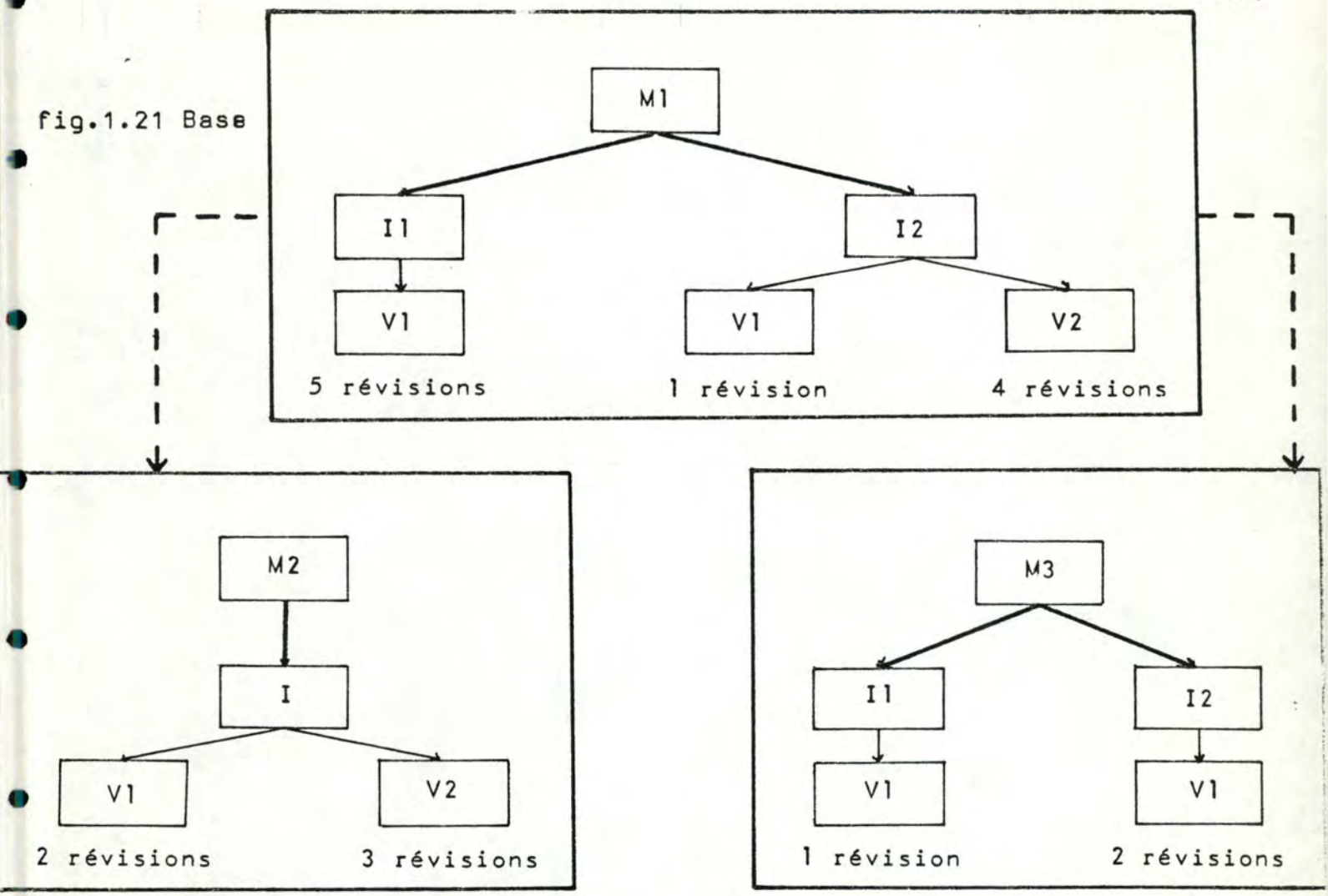
Nous allons voir au paragraphe suivant que les espaces de travail ont une importance particulière en ce qui concerne la désignation des objets.

1.3.2.6. Désignation des objets.

Tout objet peut être défini globalement, c'est-à-dire depuis la racine de la base, ou localement, c'est-à-dire par rapport à un espace de travail bien précis. Cette désignation locale présente l'avantage de limiter la longueur du nom des objets lors de leur utilisation.
Il est en effet rébarbatif de toujours encoder >F1 >F2> F6 lorsqu'on veut se référer à la famille F6. Il serait plus indiqué de la désigner simplement par F6.
L'exemple de la figure 1.20 n'est pas réellement bien choisi pour illustrer ce problème. Le lecteur pourrait en effet se demander à quoi sert une dénomination globale puisque toutes les familles possèdent un nom différent qui pourrait par conséquent les identifier. Remarquons cependant qu'il peut y avoir dans la base de multiples objets portant le même nom. F2 et F3 pourraient toutes deux contenir une version V1 et une interface I, et une désignation globale apparaît nécessaire.
Nous allons montrer dans ce paragraphe comment réduire cette nécessité, en explicitant différentes notations sur un exemple.

Soit la base suivante:

fig.1.21 Base



- - →: dépend de ——→: est réalisé par ⟶: présente

Supposons qu'un espace de travail est associé à chaque famille:

M1 détermine l'espace 1
M2 détermine l'espace 2
M3 détermine l'espace 3

Voici quelques exemples de dénomination globale:

>M1>M2	désigne la famille M2
>M1>M2-I	désigne l'interface I de M2
>M1>M2-I-V1	désigne le corps V1 de l'interface I

Voici quelques exemples de dénomination locale:

- par rapport à l'espace 1:

-I1	désigne l'interface >M1-I1
-I2-V2	désigne le corps >M1-I2-V2
M2-I	désigne l'interface >M1>M2-I
M3-I-V1	désigne le corps >M1>M3-I-V1

- par rapport à l'espace 2:

-I	désigne l'interface M1>M2-I
-I2-V2	désigne le corps M1>M3-I2-V2

Pour simplifier la désignation des objets, il existe un mécanisme de dénomination par défaut.
Par rapport à l'espace 3, on peut par exemple utiliser les notations:

-	qui désigne l'interface par défaut de M3 (la plus récemment créée), soit I2
--	qui désigne le corps par défaut de I2
>M1-I2	qui désigne le corps par défaut de I2 de M1

Remarquons qu'un nom de corps ou d'interface peut comprendre 15 caractères et que, par conséquent, les gains occasionnés par ces dénominations par défaut peuvent s'avérer appréciables.

Les différents types d'objets associés à ces familles, corps et interfaces se désignent en concaténant le nom du type précédé d'un point au nom de l'objet.
>M1>M2-I-V1.man, >M1>M2-I-V1.hist, >M1>M2-I-V1.doc et >M1>M2-I-V1.text.002 désignent respectivement le manuel, l'historique, la documentation et la deuxième révision de V1.

Nous donnerons en annexe une description formelle de ces désignations suivant la notation Backus-Naur Form.

1.3.3. Concept de configuration.

1.3.3.1. Définition.

Une configuration est l'ensemble des modules concourant à la réalisation d'une interface. Nous avons vu (cfr. 1.3.2.1.) qu'un corps C ne pouvait pas toujours à lui seul réaliser toutes les ressources de son interface I et qu'il doit parfois utiliser des ressources extérieures réalisées par d'autres corps Cj, et récursivement.
L'ensemble du corps C et des corps Cj constitue une configuration I, encore appelée réalisation composée de I.
Il s'agit en fait de la fermeture transitive du graphe de dépendance. Cette fermeture peut être obtenue à partir de n'importe quel sommet différent de feuilles. Cela induit donc la possibilité de créer des configurations à n'importe quel niveau (cfr. 1.3.2.3.) du graphe, hormis le niveau le plus élevé. Cette configuration peut donc être considérée comme un nouveau corps de l'interface I. Elle se désigne par conséquent comme un corps. Un manuel, un texte, un historique et une documentation y sont associés. Elle diffère des corps traditionnels par le contenu de son texte. Celui-ci se subdivise en deux parties:

- différentes commandes de sélection des corps devant intervenir dans sa composition. Des exemples de ces commandes seront donnés en 1.3.4. Elles seront à la charge de l'utilisateur.

- sa liste de composition remplie automatiquement lors de l'exécution d'une commande interactive de constitution de la configuration.

1.3.3.2. Exemple.

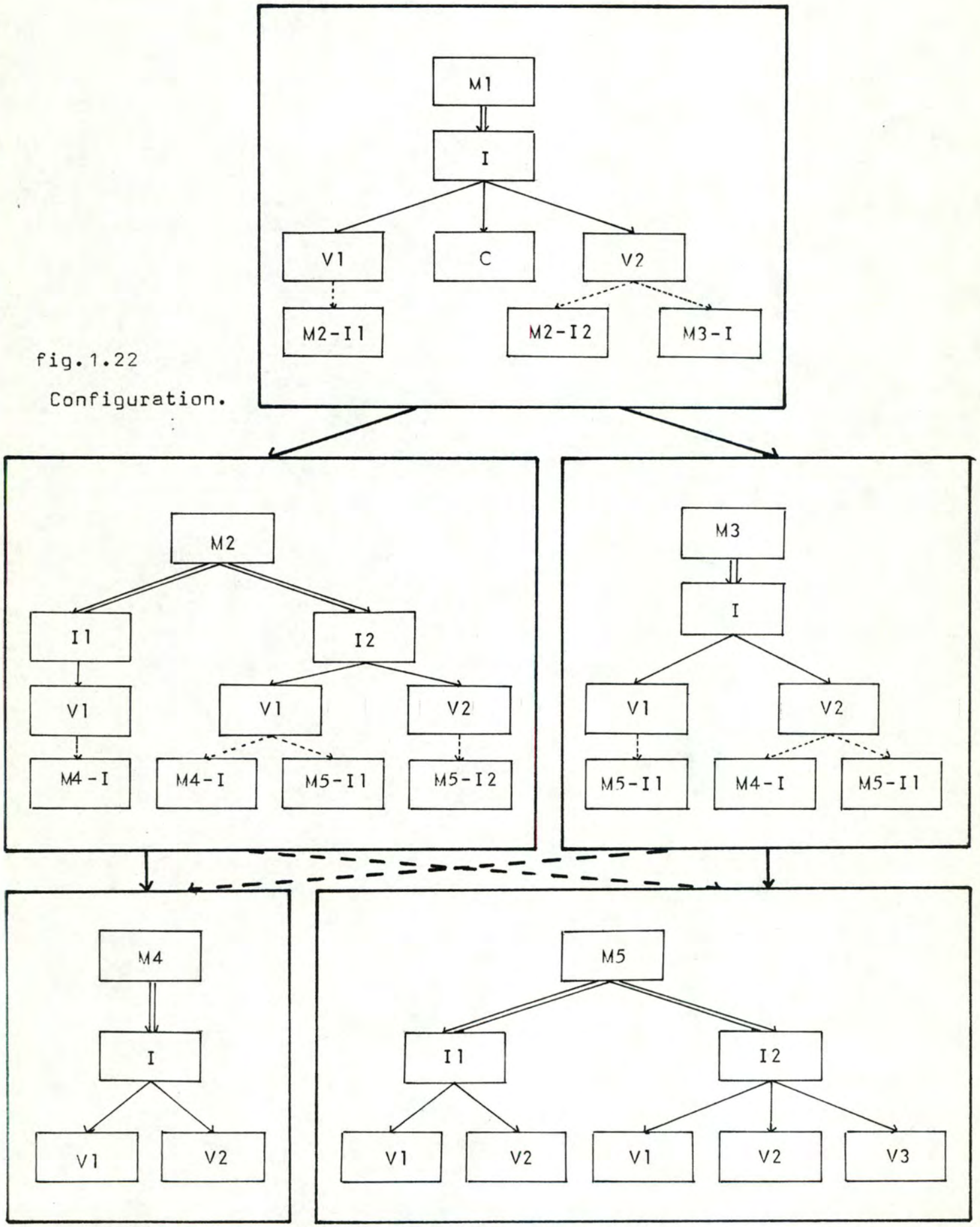


fig. 1.22
Configuration.

Soit la configuration C associée à l'interface I du module M1.
On devra pour la constituer:

1. choisir une version de M1;
2. Si V1 est choisie, - choisir une réalisation de M2-I1 (M2-I1-V1)
   - choisir une réalisation de M4-I (M4-I-V1 ou M4-I-V2)
3. Si V2 est choisie, - choisir une réalisation de M2-I2 et de M3-I

   ⋮

   etc.

A l'issue de la sélection et si les dépendances ont été exactement mentionnées par l'utilisateur, nous aurons une liste de composition de M1-I-C. Notons qu'il n'est pas exclu qu'une configuration en contienne une autre dans sa liste de composition puisque la base ne fait pas la différence entre une configuration et un corps traditionnel.
Le but du paragraphe suivant est d'expliquer comment les différents choix sont effectués.

1.3.4. Constitution d'une configuration.

Nous avons vu au paragraphe précédent que le texte d'une configuration contenait entre autres certaines commandes introduites par l'utilisateur. Ces commandes vont guider la sélection des corps intervenant dans la composition de la configuration.

1.3.4.1. Constitution manuelle.

L'utilisateur indique explicitement dans le texte de la configuration, la liste des corps intervenant dans sa composition. Il remplit par conséquent la rubrique SELIMP (sélection imposée) de la configuration avec les noms des corps choisis.

Exemple:

```
configuration   M1-I-C

    selimp      M1-I-V1
                M2-I1-V1
                M4-I-V1

end;
```

On comprend très vite que cette opération devient très coûteuse pour les configurations devant contenir des centaines de modules. C'est pourquoi, la base ADELE a prévu des mécanismes plus simples de constitution.

1.3.4.2. Constitution automatique.

C'est le cas le plus trivial. En effet, aucune commande de sélection n'y est donnée par l'utilisateur. Celui-ci suppose que le système pourra lui-même explorer le

graphe de dépendances et choisir la bonne réalisation pour chaque module rencontré. On comprend dès lors que ce procédé n'est réellement applicable que pour les logiciels où toute interface est réalisée par un seul corps. Dans d'autres cas, aucun contrôle ne peut être apporté sur les choix effectués par le système de gestion de la base. Notons que si le système doit être à même d'explorer le graphe de dépendances, il doit le connaître. C'est ainsi que la liste des dépendances d'un corps est reprise dans son manuel.
Le manuel de M1-I-V2 (fig.1.22), par exemple, contiendra:

manuel M1-I-V2

liste-dépendance: M2-I2 M3-I

D'autre part, pour pouvoir choisir le corps associé à l'interface de la configuration, il sera fait référence au manuel de cette interface qui contient, entre autres, la liste des corps associés. Le manuel de M2-I2, par exemple, contiendra:

manuel M2-I2

liste-corps: V1 V2

Remarquons finalement que l'usage de ce procédé sur l'exemple de la fig.1.22 risque de conduire à une configuration incohérente. Plusieurs interfaces sont en effet réalisées par plusieurs versions, et il est par conséquent possible que le système choisisse deux versions incompatibles au sein de deux modules en relation. C'est pourquoi la base ADELE a prévu un mécanisme de contrôle de sélection.

1.3.4.3. Constitution contrôlée.

Pour éviter les inconvénients posés par les deux autres types de constitution, la base ADELE a introduit un mécanisme de désignation par attributs. A chaque version peuvent en effet être associés plusieurs attributs de la forme "type = valeur". Ils sont fournis par l'utilisateur et consignés dans le manuel de la version.
Le manuel de M2-I2-V1 pourrait par exemple contenir:

```
manuel   M2-I2-V1

    attributs
            auteur = pierre
            type = debug
            état = experimental
            syst = MR9
```

Différents mécanismes de sélection basés sur cette désignation par attributs permettent de contrôler la constitution d'une configuration. Ces mécanismes existent sous trois formes: la sélection impérative, conditionnelle et facultative.
Nous allons expliciter chacune d'elles au moyen d'exemples.

a) Sélection impérative.

Par ce type de sélection, l'utilisateur impose que les réalisations comprenant un ou plusieurs attributs spécifiques soient choisies.

Exemple:

```
configuration   M1-I-C

    selimp   (1)    >*  (type = debug)
             (2)     OR (>M1>*  (syst = MR9)
                        >M1>*  (syst = MR10))
```

selnon (3) >* (etat = incoherent)

En analysant ces différentes commandes, le système comprend qu'il faut choisir une réalisation de M1-I qui possède l'attribut "type = debug" (1) et pas l'attribut "etat = incoherent" (3). Pour les réalisations sous-jacentes dont dépend celle choisie dans M1-I, il conviendra de sélectionner celles possédant les attributs "type = debug" (1), "syst = MR9 ou MR10" (2), et ne comprenant pas "etat = incoherent" (3). Chaque ligne des rubriques SELIMP et SELNON constitue donc une contrainte à respecter. Il n'y a aucun ordre bien précis d'évaluation des contraintes. L'analyse de celles-ci produit une liste de règles ordonnées par niveau hiérarchique dans le graphe de dépendances (d'abord M1, ensuite M2 et M3, et enfin M4 et M5).
Si on désire constituer une configuration officielle de M1-I antérieure au 10 juillet 83, on pourra écrire dans le texte de la configuration associée:

selimp >* (etat = officiel date<83-07-10)

Outre le symbole d'égalité, les signes "<", ">" et "^" sont autorisés entre un attribut et sa valeur, signifiant respectivement inférieur, supérieur et différent.
Etant donné que toutes ces commandes doivent être respectées, il pourrait arriver qu'il soit impossible de constituer la configuration. Une famille pourrait en effet ne pas comprendre de réalisation satisfaisant les contraintes. L'utilisateur est alors averti lors de la constitution et la configuration est marquée incomplète. D'autre part, il est possible que les contraintes fournies par l'usager soient incompatibles. La configuration serait dans ce cas notée incohérente.
Nous dirons également qu'une configuration est incompatible si elle renferme deux composants différents d'une même famille. Cela signifierait en effet que la configuration comprend deux versions d'un même module.

b) Sélection conditionnelle.

Afin d'éviter l'échec de certaines constitutions, on introduit ce nouveau type de contrainte pour exprimer qu'une contrainte n'est impérative pour une famille donnée que si au moins une réalisation de cette famille satisfait cette contrainte. On introduit à cette fin une nouvelle clause de sélection dans le texte des configurations, la clause SELCOND.

Supposons que l'utilisateur désire créer une configuration telle que les réalisations qui la composent comprennent les attributs "syst = MR9 ou MR10" et pas l'attribut "etat = incoherent".
Supposons également qu'il a une préférence pour les réalisations comprenant "type = debug" mais qu'il ne voit pas d'inconvénient majeur à ce qu'une réalisation ne possédant pas cet attribut soit choisie. Il n'impose pas cet attribut mais s'il existe pour une réalisation d'une famille, c'est cette réalisation qui devra être choisie.
L'attribut "type = debug" apparaît donc comme étant préférentiel ou conditionnel, l'utilisateur pourra écrire:

```
selimp
        OR (>* (syst = MR9)
            >* (syst = MR10))

selnon
        >* (etat = incoherent)

selcond
        >* (type = debug)
```

Il sera ainsi assuré que la constitution, parfois très coûteuse, n'échouerait pas si une des familles explorées ne renfermait aucune réalisation ayant l'attribut "type = debug".

Ce procédé paraît donc plus souple, mais il ne réduit pas à néant les risques d'échec.
En effet, supposons que pour une famille donnée une réalisation (V1) possède l'attribut "syst = MR10" et une autre (V2), l'attribut "type = debug". D'après la règle énoncée, le système doit choisir V1 puisque "syst = MR10" est imposé et V2 puisque "type = debug" est un attribut préférentiel. Il se trouve par conséquent dans l'impossibilité de faire un choix satisfaisant l'ensemble des contraintes. C'est la raison pour laquelle la base a introduit un dernier type de sélection, la sélection facultative.

c) Sélection facultative.

Les contraintes indiquées par attribut facultatif ne sont prises en considération que si elles n'empêchent pas de faire un choix. Elles sont indiquées dans la clause SELDEF du texte des configurations. Pour éviter le problème évoqué dans l'exemple précédent, il suffirait d'écrire:

```
selimp
        OR ( >* (syst = MR9)
             >* (syst = MR10))

selnon
        >* (etat = incoherent)

seldef
        >* (type = debug)
```

La contrainte exprimée sous la clause SELDEF signifie qu'il est préférable de choisir une réalisation possédant l'attribut "type = debug". Toutefois, si cette contrainte empêche le système de faire un choix, elle sera simplement ignorée.

1.3.4.4. Remarques.

a) De telles configurations peuvent être créées à tous les niveaux de l'arborescence. Ainsi, dans l'exemple de la figure1.22, une configuration C1 pourrait être associée à l'interface I1 de M2. Il est évident que les règles de sélection pour une configuration donnée ne pourront porter que sur des familles de niveau supérieur à celui de la configuration. En termes de la relation de dépendance, cela signifie que les contraintes ne peuvent porter que sur des familles dont dépend la famille comprenant la configuration. Dans les cas contraires, l'utilisateur serait averti de son erreur et les contraintes incorrectes seraient ignorées.

b) Toute réalisation de la base peut indiquer des contraintes. Elles sont reprises sous les rubriques de sélections (selimp, selnon, selcond, seldef) de leur manuel. Il se peut en effet que le choix d'une réalisation particulière lors de la constitution d'une configuration comporte des implications sur les choix des réalisations suivantes. Ces implications ne sont pas toujours consignées dans le texte de la configuration et ne sont, par ailleurs, souvent connues que par la réalisation elle-même.
Nous avons vu qu'une configuration pourrait "préférer" certaines réalisations à d'autres, mais si elles sont réellement choisies, il est impératif qu'elles puissent imposer de nouvelles contraintes. Cette possibilité permet en fait une décentralisation des contraintes de sélection à travers toute la base.

c) Nous avons vu (cfr 1.3.3.1.) qu'une configuration, tout comme un corps traditionnel, étaient des réalisations d'interface. Pour le système, ils sont indiscernables. Rien n'empêche par conséquent qu'une configuration soit un constituant d'une configuration englobante. Sur l'exemple de la figure1.22,une configuration M2-I2-C1 pourrait appartenir

à la liste de composition de la configuration M1-I-C.
On s'aperçoit d'autre part que M2-I2-C1 et M1-I-C sont toutes deux susceptibles de contenir une version de M5-I1.
Supposons en effet la situation suivante:

C1) on construit la configuration M2-I2-C1
en choisissant M2-I2-V1, M4-I-V1, M5-I1-V1

C2) on construit la configuration M1-I-C
en choisissant M1-I-V2, M2-I2-C1, M3-I-V1, M5-I1-V2.

On remarque par cet exemple que la configuration M1-I-C est incompatible. Elle contient en effet deux versions différentes de M5-I1 (M5-I1-V1 via M2-I2-C1 et M5-I1-V2).
La base ADELE règle ce type de conflit en retardant le choix pour M5-I1 lors de la constitution de M2-I2-C1. Le résultat de cette constitution sera une liste de composition partielle (M2-I2-V1, M4-I-V1) et la configuration héritera des contraintes de M2-I2-V1 sur M5-I1 et d'une dépendance sur cette même interface. Ainsi, lors de la constitution de M1-I-C, si M2-I2-C est choisie, on disposera de toutes les contraintes pour effectuer le choix de la réalisation de M5-I1.

1.3.5. Concept de cohérence.

1.3.5.1. Définitions.

Nous avons vu en 1.3.2.1. que la base gérait certains types de relations telles que les relations de réalisation et d'utilisation entre un corps et une interface. Ces relations peuvent être compromises par la modification d'un de leurs composants. Pour que la base reste dans un état cohérent, il est dès lors nécessaire que l'ensemble des relations reste "sincère". Il serait en effet aberrant d'affirmer qu'un corps réalise une interface modifiée qui présente les procédures A, B, C, si ce corps réalise encore les anciennes procédures X, Y et Z.
D'autre part, nous avons vu en 1.3.3.1. qu'une configuration était composée d'un ensemble de corps. Si un de ces corps est modifié, il est bien évident que la configuration qui l'inclut peut s'en trouver altérée.
Afin de favoriser la cohérence de la base, on associe un état à chaque objet. Cet état est représentatif du contenu de l'objet. Il peut aussi caractériser la nature des relations dont il fait partie. Nous avons vu en 1.3.4.3. qu'une configuration pouvait être incohérente ou incompatible. Ces deux caractéristiques constituent deux états d'une configuration.
Par rapport à ces états, on définit trois types de cohérence:

- cohérence forte: dès qu'un objet est modifié, on veille automatiquement à ce que toutes les relations dont il est composant restent cohérentes. Lorsqu'une interface est modifiée, on peut, par exemple, recompiler tous les corps qui la réalisent et qui l'utilisent. Pour que cette opération réussisse, il faut évidemment que les corps concernés aient été modifiés conformément à la nouvelle interface.

- cohérence semi-forte: lorsqu'un objet est

modifié, on positionne son état à "modifié" et on change les états des objets altérés par cette modification. Par exemple, lorsqu'on transforme une interface, on la marque à "modifié" et on positionne l'état des corps qui la réalisent ou qui l'utilisent à "incohérent". Aucune mesure automatique de recompilation n'est entreprise. Elle ne le sera que sur demande de l'utilisateur.

- cohérence faible: lorsqu'un objet est modifié, on positionne son état à "modifié", ou on dispose d'un autre mécanisme permettant de savoir s'il a été modifié. L'état des objets en relation avec l'objet modifié reste inchangé et ne sera réévalué que sur demande de l'utilisateur. A plus forte raison, toute modification du contenu d'un objet ne pourra être dirigée que par l'utilisateur.

Le système MAKE (Fel, 79) par exemple assume une politique de cohérence faible. L'utilisateur peut, en soumettant une série de commandes au système (make file), modifier certains composants du logiciel. Il peut par exemple provoquer la recompilation automatique de tous les codes sources modifiés depuis leur dernière compilation.
Nous allons voir comment ADELE assure le maintien de la cohérence de sa base de programmes.

1.3.5.2. Maintien de la cohérence.

Pour les corps et les interfaces, la base a implémenté une stratégie de cohérence semi-forte. Elle a par conséquent défini une série d'états caractéristiques. Les états connus sont les suivants:

- vide: une interface ou une réalisation est vide si elle est créée mais qu'aucun texte n'a été fourni pour valeur initiale.

- verrouillé: un objet est verrouillé si un usager se l'est réservé en vue d'une modification.

- incohérent: une réalisation devient incohérente si elle utilise ou réalise les ressources d'une interface qui depuis lors a été modifiée. Une configuration est incohérente si elle renferme des contraintes incompatibles.

- périmé: une configuration est périmée si le texte d'une réalisation qu'elle inclut est modifié.

- incomplet: une configuration est incomplète s'il existe une interface pour laquelle aucun choix n'a pu être effectué. Une telle situation se présente lorsque les contraintes sont telles qu'elles imposent la sélection de deux réalisations différentes de la même interface.

- incompatible: une configuration est incompatible si sa liste de composition renferme deux réalisations différentes d'une même interface.

- ok: un objet est dans cet état si aucun des autres états ne s'applique.

A certains de ces états est associée la liste des objets responsables de leur valeur:

- état incohérent: liste des interfaces modifiées s'il s'agit d'une réalisation, et liste des contraintes incompatibles s'il s'agit d'une configuration.

- état incomplet: liste des interfaces pour lesquelles un choix n'a pu être fait.

- état périmé: liste des réalisations modifiées.

- état verrouillé: auteur et date du verrouillage.

Ces états et listes maintenus dans les manuels peuvent à tout moment être visualisés par l'utilisateur qui peut dès lors prendre des décisions quant à la modification des objets auxquels ils se rapportent.
D'autre part, avant la modification ou la destruction de tout objet, les effets de bord sont calculés et la liste des objets dont l'état est affecté est soumise à l'utilisateur. La commande ne sera effectivement exécutée que sur confirmation de l'usager.

On constate donc, qu'à travers ce mécanisme, l'utilisateur est responsable de la cohérence au niveau des réalisations et des interfaces. Le système lui offre simplement une aide par le maintien continuel des états et de leurs listes associées. Les concepteurs de la base ont en effet jugé inutile de recompiler automatiquement, sans les modifier, les corps réalisant ou utilisant une interface altérée.
En ce qui concerne les configurations, ils ont préféré une politique de cohérence forte. Ainsi, pour une réalisation donnée, la modification de son manuel entraîne la réanalyse de toutes les configurations englobantes. En effet, la modification du manuel peut provoquer un changement des attributs de la réalisation et il convient de vérifier si cette configuration peut toujours faire partie des configurations qui l'incluent. En outre, la modification du manuel peut entraîner le changement des contraintes de sélection propres à la réalisation. Dans ce cas, il faut vérifier si les réalisations utilisées peuvent encore faire partie des configurations incluant celle dont le manuel a été modifié. D'autre part, si une configuration est modifiée, il est nécessaire de réanalyser toutes les configurations qui l'englobent et récursivement.
Bien que cette méthode paraisse substantielle au niveau de la

cohérence, il n'en est pas moins vrai qu'elle devient rapidement coûteuse et rébarbative à l'usage. C'est pourquoi les concepteurs de la base envisagent de réduire cette cohérence forte en une cohérence semi-forte dans une version prochaine.

1.3.6. Création et modification des objets de la base.

Nous allons dans ce chapitre définir les principales commandes de la base et les expliciter par un exemple de création et manipulation d'une base.
Les paramètres entre parenthèses des différentes commandes sont facultatifs et les autres sont obligatoires.

1.3.6.1. Commandes principales.

- creerbase nombase espacefichier (idperson):
cette commande permet de créer la base de nom "nombase". Tous les objets de cette base auront un nom commençant par ">nombase" et seront placés dans l'espace fichier indiqué. Le propriétaire sera "idperson" s'il est mentionné et l'exécutant de la commande sinon.
"Nombase" est le nom de la famille racine de la base.

- creer nombase:
créer l'entité "nombase". Cette entité devient connue de la base mais aucun contenu ne lui est affecté.

- reserve nombase (fichloc):
réserver l'entité de nom "nombase". Cette entité est verrouillée suite à cette commande. Seul l'utilisateur ayant effectué cette réservation pourra modifier l'entité. Si le nom d'un fichier local (fichloc) est mentionné, l'entité réservée y est copiée. Cette commande n'est effective que si l'entité était libre (non verrouillée) avant son exécution.

- catal nombase fichloc (fichier de dépendances):
cataloguer l'objet contenu dans "fichloc", dans l'entité "nombase". Cette entité doit avoir été réservée. Si un fichier de dépendances est mentionné, il contiendra l'ensemble de dé-

pendances de l'objet "nombase" sur les autres entités de la base. Ce fichier n'est admis que si "nombase" est un corps et il ne peut contenir des dépendances que sur les familles externes ou filles de la famille incluant "nombase".
Notons également que cette commande comporte de nombreuses options permettant de cataloguer simultanément le code objet correspondant, de détruire après le catalogage tous les fichiers locaux, de sortir un listing de l'objet catalogué, de donner un commentaire...

- lire nombase (fichloc):
lire l'entité "nombase". Si "fichloc" est présent, le contenu de "nombase" y est copié, sinon ce contenu apparaît à l'écran.

- detruire nombase:
destruction de l'entité "nombase".

- crinit nombase fichloc (fichier de dépendances)
cette commande enchaîne successivement les trois commandes "creer nombase", "reserve nombase fichloc" et "catal nombase fichloc (fichier de dépendances)".

- visible nomfamille (familleorigine):
la famille "nomfamille" est visible à partir de "familleorigine". Suite à cette commande, "nomfamille" devient une famille externe (cfr. 1.3.2.3.) de "familleorigine".

- dclesp nomfamille nomesp (idperson):
déclarer l'espace de nom "nomesp". Cet espace comprendra l'ensemble des familles dont dépend "nomfamille", et sera affecté à l'usager "idperson" s'il est mentionné et à l'exécutant de la commande si non.

- cesp (nomesp):
changement d'espace. L'usager désire se mouvoir sur l'espace de nom "nomesp", ou sur son espace par défaut si "nomesp" n'est pas mentionné.

1.3.6.2. Exemple de création et de manipulation d'une base.

- creerbase M1 >udd>projet>person>mabase
  (créer la base M1)
- dclesp >M1 M1
  (créer l'espace M1. Il comprend toute la base)
- creer >M1> M2
  (créer la famille M2)
- dclesp >M1>M2 M2
  (créer l'espace M2 de famille racine M2)
- cesp M2
  (se placer sur l'espace M2)
- creer M4
  (créer la famille M4, qui devient fille de M2)
- creer M4-I
  (créer l'interface I de M4)
- creer -I
  (créer l'interface I de M2)
- creer -I-V
  (créer la version V de M2-I)
- reserve --
  (réserver M2-I-V (notation par défaut))
- catal -- M2.Pascal "M4-I"
  (cataloguer M2.Pascal dans M2-I-V avec une dépendance sur M4-I)
- cesp M1
  (se placer sur l'espace M1)
- creer M3
  (créer la famille M3 qui devient fille de M1)
- dclesp >M1>M3 M3
  (créer l'espace M3)

- cesp M3 (se placer sur l'espace M3)

- visible > M1 > M2 > M4 (la famille M4 est visible de M3. M4 devient externe à M3)

- creer -I (créer l'interface I de M3)

- crinit --V M3.Pascal "M4-I" (créer et réserver M3-I-V puis y cataloguer M3.Pascal avec une dépendance sur M4-I

A l'issue de ces différentes commandes, la base est dans l'état suivant:

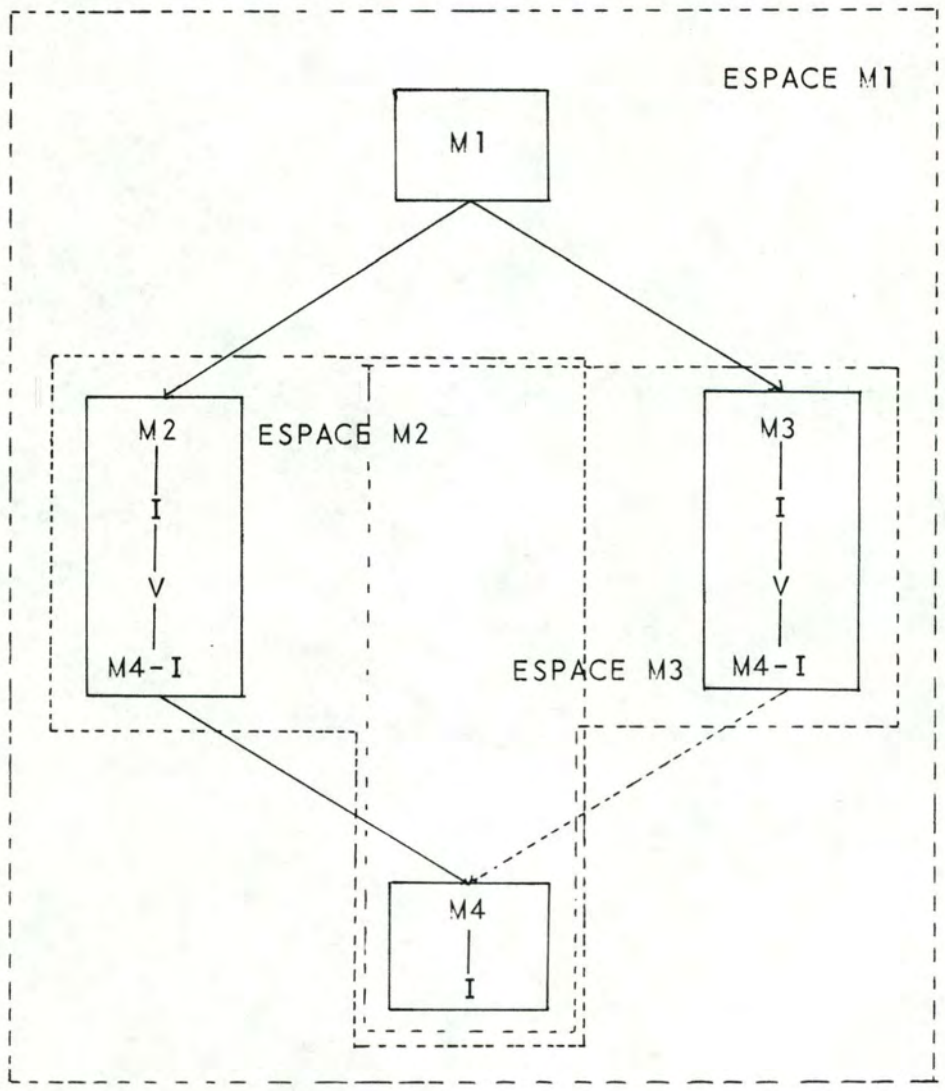


fig. 1.23 Base créée.

2. RCS: un système de gestion de révisions. (Tic, 82)

2.1. Introduction.

RCS (Revision Control System) est un outil, développé autour du système UNIX, permettant de gérer efficacement des révisions de textes. Le mot "gérer" sous-entend la conservation ou mémorisation des révisions, leur mise à disposition de l'utilisateur .... Cette gestion sera explicitée aux paragraphes suivants. Le mot "texte" étant pris dans son sens général, il peut s'agir de programmes, de documentation, ou de tout ensemble de lignes obéissant ou non à une structure bien définie. Nous entendons par ligne, toute suite de caractères différents de CRLF (Carriage Return and Line Feed) terminée par CRLF.

Notons avant d'entrer dans les détails, que bien que l'ordre de présentation de ce travail tende à laisser supposer le contraire, c'est bien ADELE qui s'est inspiré de RCS et non l'inverse.
Nous allons, au paragraphe suivant, expliciter la structuration des révisions de textes sous RCS, pour ensuite relater brièvement les principales fonctionnalités de ce système, notre but étant surtout d'étudier la façon dont est conservé l'ensemble des révisions. Nous avons en effet réalisé un travail similaire sous ADELE. Nous consacrerons par conséquent le paragraphe 2.4 à la méthode utilisée par RCS pour mémoriser ses révisions successives.

2.2. Structuration des révisions d'un texte.

Avant de parler explicitement de structure, disons simplement que RCS permet à l'utilisateur de lui soumettre des révisions et de lui en demander.
Pour qu'un utilisateur puisse soumettre une nouvelle révision à RCS, il faut qu'il ait verrouillé la précédente. Sans cette

précaution, deux utilisateurs pourraient simultanément lire et modifier le même texte. Etant donné qu'ils ont lu une même révision A, lorsqu'ils soumettront ultérieurement chacun leur copie à RCS, elles seront conservées sous la même désignation, soit B. Il est dès lors évident que celui qui soumettrait sa copie le dernier annulerait inéluctablement les modifications de l'autre.
Ce principe de verrouillage est semblable à celui opéré lors des mises à jour dans les bases de données classiques.
En travaillant de la sorte, nous aboutissons à une série de révisions successives obéissant à une structure linéaire du type:

$$1.1 \longrightarrow 1.2 \longrightarrow 1.3$$

$\longrightarrow$: précède

Notons cependant que même si une révision est verrouillée par un utilisateur A, elle peut toujours être lue par un utilisateur B. La seule opération qui lui est interdite est de resoumettre cette révision dans la lignée des révisions existantes. Supposons en effet la situation suivante:

- l'utilisateur A lit 1.3 et la verrouille ;
- l'utilisateur B lit 1.3 :
- l'utilisateur A modifie et resoumet 1.3 sous 2.1 à RCS, ce qui conduit à 1.1 --→ 1.2 --→ 1.3 --→ 2.1 ;
- l'utilisateur B veut resoumettre 1.3 à RCS. Puisqu'il ne l'a pas verrouillée, cette opération lui est interdite ;

Ce genre de situation peut toutefois s'avérer gênant. En effet, le verrouillage de l'utilisateur A peut durer longtemps et d'autre part, l'utilisateur B peut désirer que ses modifications soient indépendantes de celles de A. C'est pourquoi le système lui permet de soumettre sa révision de la manière décrite ci-dessous:

RCS offre donc la possibilité de maintenir plusieurs révisions issues d'un même ancêtre (2.1 et 1.3.1.1 issues de 1.3) .
La généralisation de cette situation conduit à la création d'une arborescence de révisions du type:

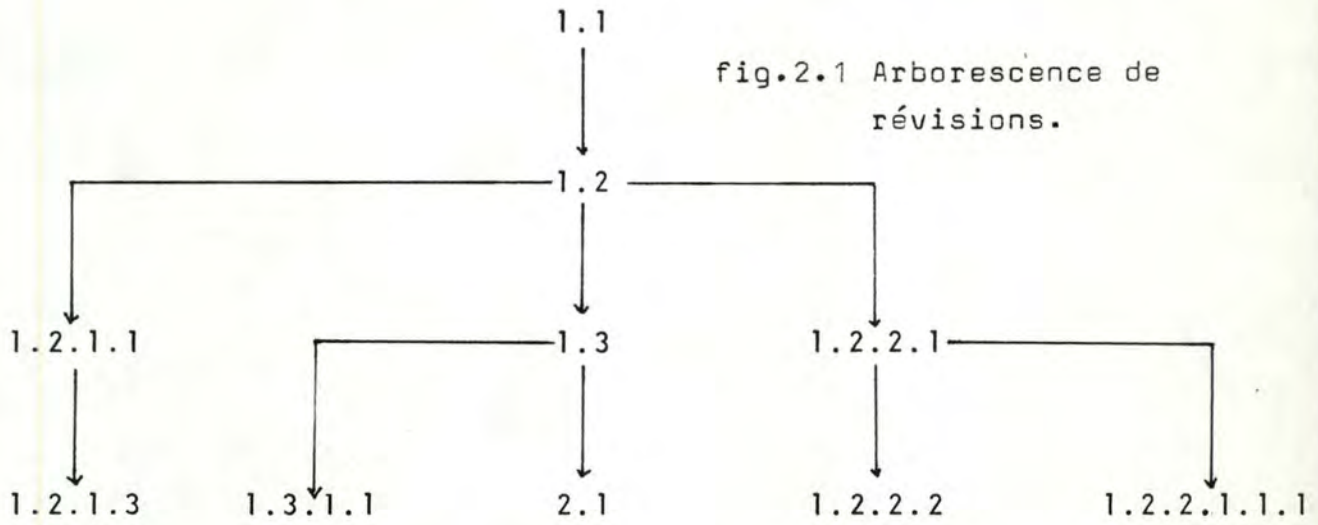


fig.2.1 Arborescence de révisions.

⟶ : précède

Pour rester cohérent avec les dénominations de RCS, nous dirons que l'axe principal de cette arborescence, à savoir 1.1 --→ 1.2 --→ 1.3 --→ 2.1, s'appelle le tronc et les axes qui en sont issus s'appellent les branches. Le tronc représente donc l'ensemble des noeuds linéaires contenant la racine, et les branches représentent tout autre ensemble de noeuds linéaires. En terme de révisions, le tronc contient toujours la première révision créée.
Notons également que RCS utilise certaines conventions de numérotation des révisions. C'est ainsi que la première révision de la première branche (dans le temps) issue d'une révision X portera le nom de X.1.1 et celle d'une seconde branche éventuelle portera le nom X.2.1 . On impose d'autre part que les numéros des révisions successives des branches ou du tronc soient dans un ordre strictement croissant.

2.3. Opérations sur l'arborescence des révisions.

Pour gérer des révisions de texte, RCS fournit dans son interface différentes commandes accessibles à l'utilisateur.

2.3.1. Introduction d'une révision de texte.

Une commande importante est celle qui permet à l'utilisateur de soumettre un texte au contrôle de RCS. Cette soumission s'opère via la commande Ci (Check in). Par exemple, l'opération "Ci X" a pour but d'intégrer le fichier X sous RCS. Si le système ne possède pas encore de fichier X sous sa gestion, il initialisera un fichier X.v dont le texte de X sera la révision 1.1 . Si par contre le fichier X.v existe déjà, le système attribuera au texte soumis un numéro directement supérieur à celui de la dernière révision soumise (p.e. 1.1, 1.2, 1.3). L'utilisateur a par ailleurs le choix de déterminer lui-même le numéro de révision pourvu que celui-ci soit supérieur au numéro de la révision précédente. Ainsi, la commande "Ci -r2 X" n'est valide que si le numéro de la révision précédente est de la forme 1.j, et elle assigne le numéro 2.1 à la nouvelle révision intégrée sous "X.v". Si l'utilisateur avait simplement frappé la commande "Ci X", le système aurait assigné le numéro 1.4 à la révision si la précédente était désignée par 1.3 .
Notons que lors d'un check in, le système enregistre la date et l' identification du programmeur auquel est demandée une description sommaire de son texte ou de la modification qu'il y a apportée.

2.3.2. Obtention d'une révision d'un texte.

L'opération inverse consiste à extraire une révision particulière. Pour cela, l'utilisateur possède la commande Co (check out). La simple commande "Co X" permet

d'extraire la dernière révision intégrée sous X.v .
Différents critères ou options permettent de déterminer d'une manière plus précise la révision à retirer. Par exemple, "Co -r2.4 X" permet d'extraire la dernière révision dont le numéro se situe entre 2.1 et 2.4. Ou encore la commande "Co -d9/12 X" permet de retirer la dernière révision soumise avant le 12 septembre de l'année courante. Il existe plusieurs autres critères de sélection comme l'auteur du check in ou l'état de la révision, mais nous n'entrerons pas dans ces détails fastidieux, le lecteur intéressé pouvant facilement se référer au manuel utilisateur de RCS.

2.3.3. Désignation symbolique d'une branche.

Pour éviter de devoir sans cesse se remémorer le numéro de la branche qui lui appartient, tout programmeur peut lui assigner un ou plusieurs noms symboliques. Si, par exemple, la branche 1.2.2 est symbolisée par "temp", la dénomination temp.1 identifiera la révision 1.2.2.1 .
D'autre part, remarquons que d'un point de vue pratique, toutes les révisions au sein d'une même arborescence sont intégrées au même fichier X.v et qu'un ensemble X1.v, X2.v ... Xn.v de ces fichiers constitue un ensemble de n modules.
Un module sous RCS est donc caractérisé par une arborescence de révisions implémentée par un fichier unique.
Pour extraire de cet ensemble de modules les n révisions qui appartiennent à une même configuration (cfr. ADELE), il suffit de nommer la branche intéressée de chaque arborescence "config" (par exemple) et d'exécuter la commande "co -rconfig *.v". Cette commande extraira les dernières révisions de toutes les branches nommées "config" de toutes les arborescences gérées par RCS. Et puisque plusieurs noms peuvent être assignés à une même branche, une même révision peut participer à plusieurs configurations.

2.3.4. Jonction de révisions.

Une fonction importante offerte par RCS et issue de l'existence de telles arborescences consiste en la jonction de deux révisions par rapport à une troisième. L'opération qui consiste à joindre les révisions r1 et r2 par rapport à r3 permet d'effectuer sur r2 les modifications qui ont conduit de r3 à r1. Cette possibilité apparaît intéressante dans le cas de modifications globales devant s'appliquer à une série de révisions. Il suffit de l'exécuter une seule fois pour ensuite la reporter automatiquement sur les révisions intéressées. Remarquons toutefois que de telles opérations peuvent devenir ambiguës. En effet, supposons les trois révisions suivantes:

```
r3
┌
(lignes début)
(lignes fin)
└
```

```
r2
┌
(lignes début)
 ligne "titi"
(lignes fin)
└
```

```
r1
┌
(lignes début)
insertion "toto"
(lignes fin)
└
```

Le report sur r2 des modifications qui ont conduit de r3 à r1 indique qu'il faut inclure la ligne "toto" après le bloc (lignes début) et avant le bloc (lignes fin) de r2 (en supposant que ces différents blocs soient identiques dans les 3 révisions).
On remarque que le report est ambigu. Faut-il insérer toto avant ou après titi ? C'est à cause de cette classe de problèmes que l'outil développé est essentiellement interactif et qu'il incombera à l'utilisateur de lever les dif-

férentes ambiguïtés.
Ces différentes explications permettent de se faire une idée générale sur RCS. Il est évident que la liste des fonctions fournies est loin d'être exhaustive et nous renvoyons une fois de plus le lecteur avide de renseignements complémentaires au manuel utilisateur.
Nous allons maintenant nous attarder à un aspect important de l'implémentation de cet outil, à savoir la technique de mémorisation des révisions successives.

2.4. Mémorisation des révisions.

2.4.1. Introduction.

Pour conserver ses différentes révisions, RCS utilise la technique des deltas. Cela signifie qu'au lieu de conserver le texte intégral de toutes les révisions, le système n'en garde qu'un seul et conserve les différences entre révisions successives.
Le grain d'atomicité de changement entre deux textes successifs est la ligne. Ainsi, si un seul caractère change dans une ligne entre deux textes A et B, RCS considère que toute la ligne a changé.
Le delta entre deux textes A et B est l'ensemble des lignes appartenant à A et non à B, et vice versa.
On se pose immédiatement à ce stade les questions de savoir quelle révision mémoriser intégralement et de quelle manière conserver les deltas entre révisions successives. C'est à ce type de questions que nous allons répondre.
Nous allons d'abord étudier une technique de deltas fusionnés utilisée par SCCS (Source Code Control System) (Roc, 75) et ensuite expliquer le choix de RCS, à savoir l'utilisation de deltas séparés, directs ou inverses.

2.4.2. Deltas fusionnés.

Cette technique d'implémentation de deltas consiste à inclure les différences à l'intérieur même de la révision d'origine. Cette inclusion s'opère à l'aide de marqueurs qui délimitent les ensembles de lignes appartenant à une révision déterminée.
Nous allons examiner cette notion sur un exemple.

Supposons qu'une révision 1.1 contienne le texte suivant:

```
1    aa
2    bb
3    bb
4    cc
5    dd
```

Cette révision soumise à SCCS deviendra:

```
1*   I1.1
2    aa
3    bb
4    bb
5    cc
6    dd
7*   E1.1
```

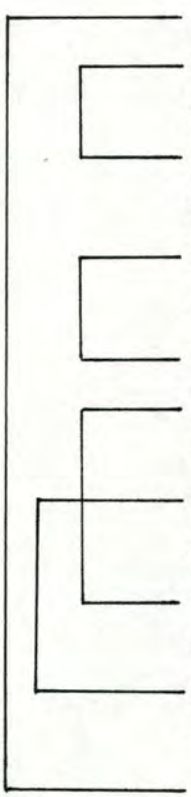
où les marqueurs I1.1 (Insert in 1.1) et E1.1 (End of 1.1) délimitent les lignes à insérer dans 1.1 .

Supposons maintenant qu'on crée la révision 1.2 en supprimant les lignes 3 et 4 de la révision 1.1 .
Le texte résultant sera:

```
1*   I1.1
2    aa
3*   D1.2
4    bb
5    bb
6*   E1.2
7    cc
8    dd
9*   E1.1
```

où les marqueurs D1.2 (Delete for 1.2) et E1.2 (End of 1.2)

délimitent les lignes à supprimer pour 1.2 .
Ce type d'implémentation peut conduire à des fichiers complexes où les modifications successives peuvent s'entrelacer.
On aboutit fréquemment à des fichiers du type suivant:



Pour régénérer une révision particulière, un programme doit balayer la totalité du texte et en sélectionner les parties appropriées. Pour régénérer une révision i.j, il suffit de prendre en considération tous les marqueurs d'insertion et de suppression portant sur les révisions de numéro inférieur à i.j.
Pour recréer la révision 1.2 de l'exemple précédent, il convient de reprendre les textes (2), (4), (5), (6), (7) et de supprimer les textes (1) et (3). La régénération de 1.3 exigera la récupération des textes (2), (4) et (7).
L'avantage de cette technique réside dans le fait qu'elle occasionne un gain de place maximum et une propriété intéressante est qu'elle prend le même temps pour régénérer n'importe quelle révision. Son inconvénient majeur provient du fait qu'à

tout catalogage d'une nouvelle révision, il est nécessaire de régénérer la dernière, de calculer la différence et de l'intégrer dans le texte. La complexité de cette opération résulte de la complexité et du manque de structure du fichier des révisions. C'est pourquoi RCS a plutôt choisi une politique d'implémentation de deltas séparés.

2.4.3. Deltas séparés.

Cette section présente la stratégie choisie par RCS pour conserver ses révisions. Elle se divise en deux paragraphes: le premier présente la stratégie en ce qui concerne le tronc de l'arborescence des révisions et le second s'attarde aux branches.
Rappelons qu'en termes d'arborescence, le tronc représente l'ensemble des noeuds linéaires qui contient la racine, tandis que les branches représentent tout autre ensemble de noeuds linéaires.

2.4.3.1. Cas du tronc.

Reprenons les deux révisions ayant servi à illustrer SCCS.

Révision 1.1	Révision 1.2
aa	aa
bb	cc
bb	dd
cc	
dd	

En supposant que la première révision soit gardée en clair, ces deux textes soumis à RCS deviendraient:

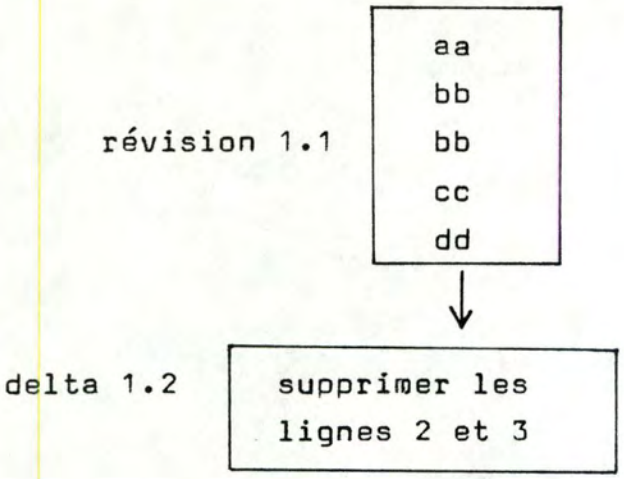


fig. 2.2 Delta.

Notons que le delta 1.2 consiste en fait en un ensemble de lignes d'édition qui, appliquées à la révision 1.1, permettent de régénérer automatiquement la révision 1.2 . Ces commandes d'édition sont simplement obtenues en appliquant la commande "diff" de UNIX aux deux textes. Un tel delta est appelé delta direct parce qu'il permet de régénérer une révision j à partir de la révision de numéro directement inférieur.
En continuant selon cette technique, nous aboutirions à des structures du type:

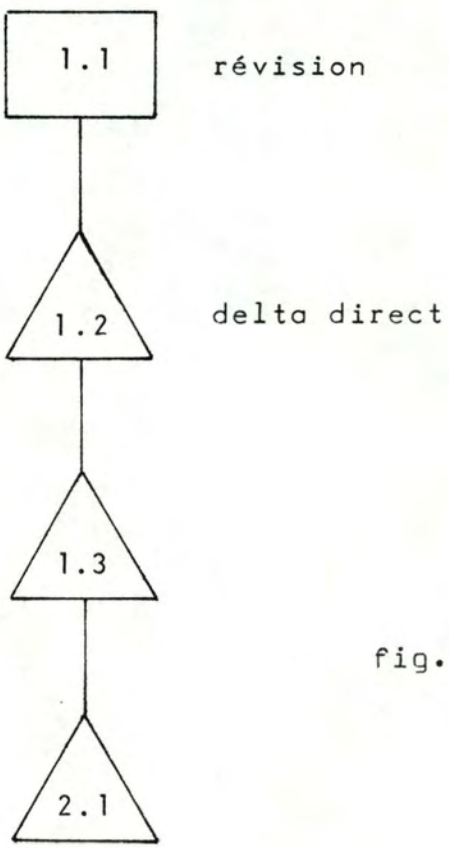


fig.2.3 Deltas directs.

Cette façon de procéder présente un inconvénient majeur: en effet, lors du catagogage de toute nouvelle révision, il faut régénérer la dernière avant de calculer la différence et d'autre part, la dernière révision est en pratique la plus souvent référencée. C'est pourquoi RCS a choisi de conserver intégralement la dernière révision plutôt que la première. La figure2.2devient alors:

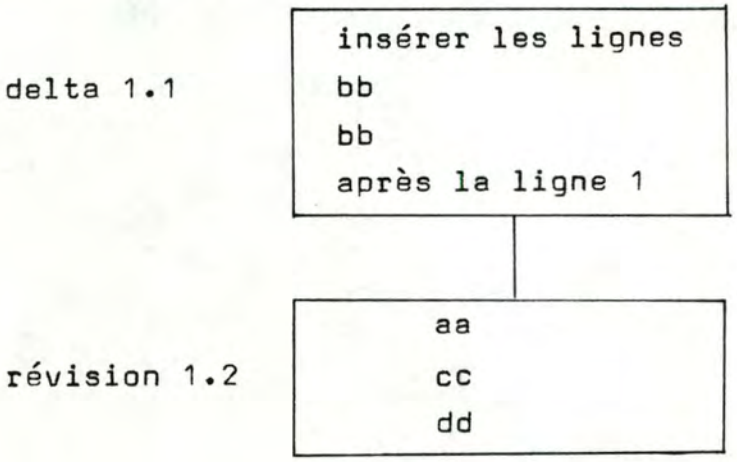


fig.2.4 Conservation de la révision 1.2.

Un tel delta est appelé <u>delta inverse</u> parce qu'il permet de régénérer une révision j à partir du texte de la révision de numéro directement supérieur. Nous aboutissons ainsi à une structure du type:

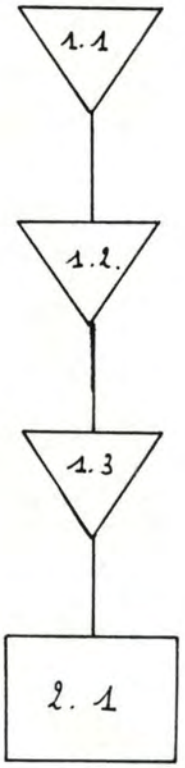


fig.2.5 Deltas inverses.

Remarquons que dans ce cas, le temps d'extraction d'une révision n'est plus le même pour toute révision, mais dépend du nombre de deltas à appliquer. La reconstitution de la révision 1.1 provoquera la reconstitution de deux révisions intermédiaires.

2.4.3.2. Cas des branches.

En ce qui concerne les branches, RCS a choisi d'implémenter des deltas directs par rapport à la révision dont elles sont issues. Cela permet de toujours construire des deltas entre révisions successives et de n'avoir qu'un type de deltas par branche. Les concepteurs de RCS auraient (ou ont) peut-être pu imaginer de construire un delta direct entre la révision dont est issue une branche et la dernière révision de celle-ci, pour ensuite construire des deltas inverses entre cette révision et les révisions précédentes de la branche.
Cette solution blesse toutefois par un manque d'homogénéité d'une part, et par une perte d'efficacité d'autre part, du fait que le delta direct construit relie deux révisions non successives.
Nous trouvons pour notre part que la solution choisie par RCS est une solution très honnête.
En reprenant les notions de deltas, la figure2.1 devient donc:

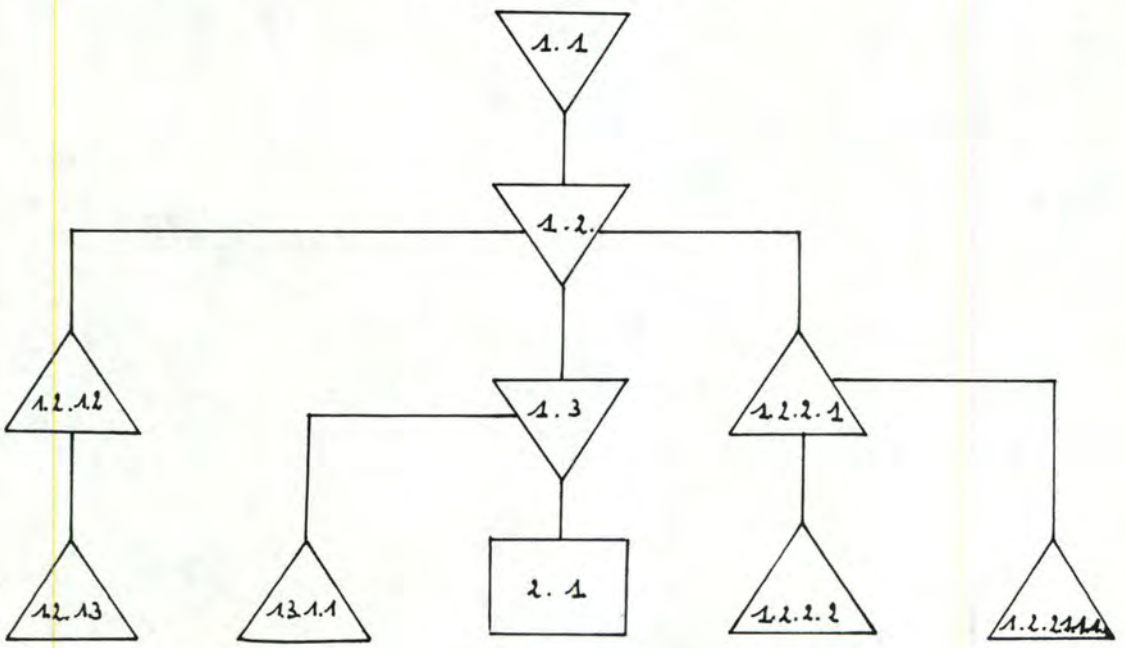


fig.2.6 Arborescence de révisions.

Notons que la manière de numéroter ses révisions permet à RCS de retrouver facilement le chemin à suivre pour regénérer la révision d'une branche. Pour reconstituer le texte 1.2.2.1.1.1 il faudra regénérer 1.2.2.1 qui à son tour nécessite la reconstitution de 1.2 . Ces différents numéros s'obtiennent facilement en éliminant successivement les deux chiffres de droite jusqu'au moment où l'on obtient un numéro ne contenant plus que deux chiffres.
D'un point de vue pratique, notons finalement que tous ces deltas et révisions sont consignés dans un seul et même fichier.

DEUXIEME PARTIE : intégration des fonctions de mémorisations de versions et de gestion d'historiques.

1. Un système de gestion d'historiques.

1.1. Première spécification.

1.1.1. Objectifs.

Nous avons vu dans la première partie au paragraphe 1.3.2.5. que la base ADELE est constituée d'une variété d'objets. C'est en effet une base relationnelle qui se compose d'entités telles que familles, interfaces, réalisations d'interfaces (corps), manuels, documents ..., et d'associations entre ces entités. En ce qui nous concerne, il importe de savoir que:

à une famille sont associés
- une documentation
- un manuel

à une interface sont associés
- un texte
- une documentation
- un manuel

à une réalisation sont associés
- des textes
- des codes objets
- une documentation
- un manuel

Si nous reprenons ceci sur un exemple de réalisation, soit F-I1-V1 où F est la famille, I1 l'interface et V1 le corps, nous avons:

objet	objet associé	type	dénomination
F	MAN	manuel	F.man
	DOC	documentation	F.doc
F-I1	MAN	manuel	F-I1.man
	DOC	documentation	F-I1.doc
	TEXTE	texte	F-I1.text
F-I1-V1	MAN	manuel	F-I1-V1.man
	Spec. test	documentation	F-I1-V1.doc
	1 2 3	texte	F-I1-V1.text.02
	MULTICS	code objet	F-I1-V1.multics
	VAX	code objet	F-I1-V1.vax

A partir d'une révision de la version nommée F-I1-V1, nous pouvons créer une nouvelle version, soit F-I1-V2. D'une révision de cette dernière, ou de la première, il est encore possible de créer d'autres versions et récursivement. Nous constatons donc qu'une interface peut posséder plusieurs versions de réalisations qui sont liées entre elles. Schématiquement, cela conduit à des réalisations du type suivant:

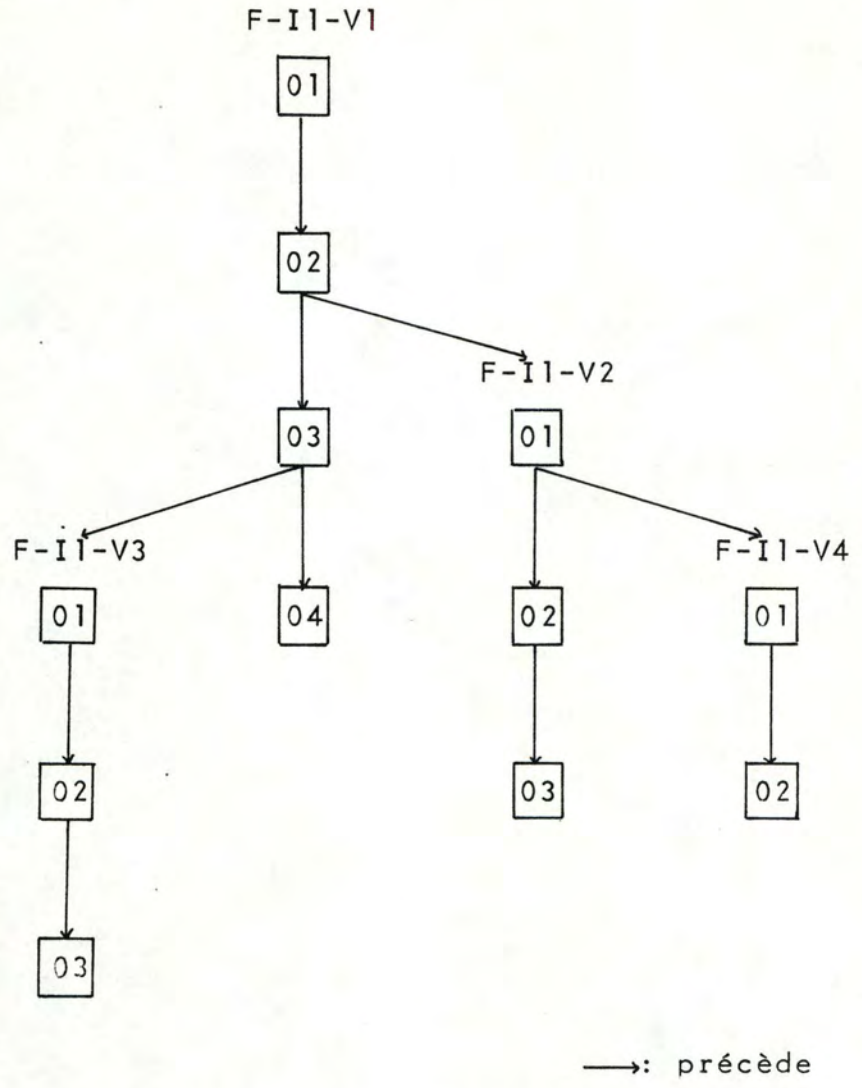


fig.1.1 Réalisations liées.

On remarquera sur ce schéma que:

F-I1-V2.01 est née de la modification de F-I1-V1.02
F-I1-V3.01 ——————————— F-I1-V1.03
F-I1-V4.01 ——————————— F-I1-V2.02

Il est intéressant de garder une trace de ces liens pour permettre à l'utilisateur de comprendre l'historique de formation d'une version ou d'une révision.
D'autre part, plusieurs utilisateurs sont susceptibles de travailler sur une même version et par conséquent de contribuer au développement de cette arborescence.
Dans un tel contexte, il est avantageux de savoir à tout moment qui a fait quoi, quand et pourquoi.

C'est dans cette perspective que nous voulons associer à toute réalisation un nouveau type d'objet, le type historique, au même titre qu'à une réalisation sont déjà associés un manuel, une documentation, des textes et des codes objets. Si nous n'avons parlé jusqu'à présent que de l'évolution des textes d'une réalisation, il ne faut cependant pas perdre de vue que les autres types associés (doc, manuel, codes objets) évoluent également. Pour que l'historique soit complet, il devra se référer à l'évolution de tous les types d'objets associés à chaque réalisation.
Etant donné que l'historique retrace l'évolution des objets associés à une réalisation, il nous a paru cohérent de garder de la même manière une trace de l'évolution du manuel et de la documentation d'une famille, et du manuel, de la documentation et du texte d'une interface.
Nous avons par conséquent généralisé le concept d'historique aux différents types d'entités gérés par la base, à savoir les familles, les interfaces et les réalisations.

D'autre part, quand l'utilisateur travaille sur le texte d'une réalisation, il est intéressant qu'il ait des informations pertinentes sous les yeux. Il peut par exemple se demander quel est l'auteur de la dernière modification d'un texte, à quelle date elle a été faite, quelles sont les caractéristiques de la version sur laquelle il travaille .... C'est à ce type de question qu'il doit pouvoir répondre rapidement.
Dans cet objectif, nous avons inséré différentes informations à l'intérieur du texte des réalisations.

1.1.2. Définition des fonctionnalités du système.

Ce chapitre est destiné à la définition des différentes fonctions nécessaires à la gestion des historiques. Nous verrons que la gestion d'un historique est décomposée en deux parties principales:

- une gestion locale à l'intérieur même du texte des réalisations;
- une gestion globale, maintenue dans un objet séparé. Celle-ci pourra être visualisée à l'aide d'une commande spécialement conçue.

Ces différents concepts, à savoir la gestion locale ou insertion d'informations, la gestion globale et la visualisation font l'objet des paragraphes ci-après.

1.1.2.1. Insertion d'informations dans le texte des réalisations.

Chaque fois que le texte d'une réalisation est catalogué par l'utilisateur dans la base, nous y insérons différentes informations jugées intéressantes et utiles lors des éventuelles lectures ultérieures. Ces informations reprennent la date et l'auteur du catalogage, le nom complet du corps catalogué, et la liste des attributs du manuel qui y sont associés au moment du catalogage.

Ces deux derniers renseignements méritent qu'on s'y arrête un instant. A quoi cela sert-il de conserver le nom complet d'une réalisation alors que l'utilisateur vient de la lire et qu'il peut par conséquent encore facilement se le remémorer ?
Il ne faut, à cet égard, pas oublier qu'un utilisateur peut lire une réalisation, c'est-à-dire l'extraire de la base pour l'amener dans son propre répertoire et la conserver un

temps quelconque avant de la recataloguer. Dans cette perspective, il est bien clair qu'un utilisateur conservant sa réalisation plusieurs jours peut rapidement en oublier le nom. Dans ces conditions, il ne possède aucun recours pour pouvoir déterminer sur quelle version et révision il travaille. En effet, le nom du fichier local est tout à fait quelconque et le nom de son programme équivaut au nom du module, c'est-à-dire au nom local de la famille. C'est pour cette raison que l'insertion du nom complet dans le texte d'une réalisation nous a paru pertinente.
D'autre part, au sein d'un même module, de multiples versions peuvent coexister et la spécificité d'une version par rapport à une autre n'est pas toujours mise en évidence par son nom. C'est pourquoi nous avons décidé d'inclure les attributs de la réalisation. Cette insertion et celle du nom complet permettront à l'utilisateur de déterminer très vite et sans ambiguïté sur quelle version d'un module il travaille.

Nous avons d'autre part désiré que l'utilisateur puisse garder à l'intérieur du texte de ses réalisations une trace de leur évolution dans le temps, appelée journal. C'est ainsi que nous pouvons maintenir, si l'utilisateur le désire, un journal interne relatant la vie de la réalisation à travers l'évolution de son développement. Cette trace peut être complète si l'usager a manifesté son intention de la posséder dès la naissance de l'objet, ou partielle s'il n'a manifesté cette volonté qu'à un moment ultérieur de son développement.
Elle contiendra des informations concernant les différentes réalisations intermédiaires ayant conduit à la création de la réalisation concernée. Ces informations engloberont l'auteur et la date de leur création ainsi que leur nom complet.

L'insertion de ces informations est guidée par l'utilisation de différents mots-clés. Ils sont introduits par l'usager aux différents endroits du texte où il désire voir apparaître les informations. La gestion de ces mots-clés est explicitée au paragraphe suivant.

1.1.2.1.1. Définition des mots-clés.

a) Leur structure.

Un mot-clé est une suite de caractères délimitée de part et d'autre part le symbole "%". On peut le rencontrer sous une des formes suivantes: %mot-clé% ou %mot-clé : chaîne quelconque %.

b) Les opérations les concernant.

Un mot-clé est inséré par l'utilisateur à l'intérieur du texte des réalisations. Cette insertion est opérée là où il désire voir apparaître les informations définies en 1.1.2. Un mot-clé est associé à chaque information et rien n'empêche qu'il soit inséré à plusieurs endroits du texte de la réalisation. Il peut également être inséré à n'importe quel endroit d'une ligne et plusieurs mots-clés peuvent se succéder sur une même ligne. Dans ce cas, ils pourront être séparés par un seul caractère "%". Par exemple, la chaîne "%mot-clé1%%mot-clé2% est équivalente à "%mot-clé1%mot-clé2% du point de vue du système.

D'autre part, un mot-clé inséré par l'utilisateur est remplacé par le système lors du catalogage de la réalisation qui le contient. Chaque fois que le système rencontre une occurence de %mot-clé% ou %mot-clé : chaîne quelconque %, il la remplace par %mot-clé : valeur % où "valeur" dépend du mot-clé rencontré.

c) Les mots-clés utilisés dans notre contexte et leur sémantique.

Afin de simplifier l'explication, nous ne reprenons les différents mots-clés que sous leur première forme,

à savoir %mot-clé%.
Nous allons les reprendre un à un et expliquer leur remplacement.

- %objet% sera remplacé par %objet : valeur % où valeur représente le nom complet de l'objet à cataloguer.

- %auteur% sera remplacé par %auteur : valeur % où valeur est le nom système de l'auteur du catalogage (ex.: Estublier.ADL).

- %date% sera remplacé par %date : AA-MM-JJ..hhmn %
  où AA = année
  MM = mois
  JJ = jour
  hh = heure
  mn = minute
  } du catalogage

- %attribut% sera remplacé par %attribut : attr1=val1: attr2=val2 ; ... ; attrn=valn % où les différents couples attributs-valeurs appartiennent au manuel associé à la réalisation au moment du catalogage. La liste d'attributs étant à priori indéterminée, nous avons veillé à n'en pas mettre trop sur une même ligne afin de garder une bonne lisibilité des listings. C'est ainsi que le nombre de couples attributs-valeurs est limité à raison de 60 caractères par ligne. Les attributs excessifs seront insérés sur la ligne suivante et précédés du mot-clé %attribut(suite). On aura par conséquent toujours une ligne de "premiers" attributs et un nombre quelconque, éventuellement nul, de lignes d'attributs "suivants".

- %journal% est un mot-clé quelque peu différent des autres par son mode de remplacement. Chaque fois que le système rencontre une occurence de ce mot-clé dans

une ligne quelconque du texte, il insère une ligne d'informations en fin de cette ligne. Elle contiendra la date et l'auteur du catalogage, ainsi que le nom de l'objet catalogué. Nous avons vu précédemment que ces trois informations pouvaient remplacer les mots-clés %date%, %auteur% et %objet%. La différence dans le cas du journal provient du fait que ces trois informations viennent compléter une liste existante, éventuellement vide. Ce procédé permet donc en quelque sorte de maintenir un journal de bord à l'intérieur d'une réalisation en relatant son évolution chronologique.
On aura une structure du type:

```
        %journal%
(1)     auteur - date - nomobjet ectuel
        auteur - date - nomobjet
                    :
        auteur - date - nomobjet
```

où les lignes successivement insérées apparaissent par ordre décroissant de dates.
La ligne (1) reprend les mêmes valeurs que celles qui remplacent %auteur%, %date%, et %objet%.

d) Remarques.

- Les informations concernant l'auteur, la date, le nom de l'objet et les attributs doivent être présentes dans toute réalisation.
Les premières lignes d'une réalisation auront toujours la structure suivante:

```
%auteur : ... %date : ... %objet : ... %
%attribut : ... %
{ %attribut(suite) : ... %
  :
  %attribut(suite) : ... % }
```

Les lignes des mots-clés correspondant à ces informations seront insérées automatiquement en début de texte.

- Les lignes insérées par le système dans le texte des réalisations doivent être entourées de délimiteurs de commentaires. Cette précaution est évidemment nécessaire pour éviter d'éventuelles erreurs de compilation ultérieures. Ces délimiteurs varient en fonction du langage dans lequel est écrite la réalisation. Ils vaudront (* et *) s'il s'agit du Pascal, /* et */ s'il s'agit du Pl1, c si le langage est Fortran...
Les lignes touchées principalement par cette remarque sont celles insérées en début du texte, les lignes d'attributs suivants et celles du journal. Il est clair que cette précaution reste à l'initiative de l'utilisateur pour les mots-clés qu'il insère lui-même.

- Le journal ne retracera l'évolution d'une réalisation qu'à partir du moment où le mot-clé correspondant est inséré dans le texte de cette réalisation. Certains pourraient en effet croire que le mot-clé %journal a un effet rétroactif et qu'il permet de retracer toute l'évolution d'un objet, indépendamment du moment de son insertion.

1.1.2.2. Gestion d'un historique séparé.

La forme d'historique dont nous venons de parler au paragraphe précédent apparaît très spécifique à différents points de vue.
Nous pouvons en effet remarquer que:

- ce type d'historique est en fait très restrictif. Il ne se rapporte qu'au texte des réalisations alors que la base ADELE comporte plusieurs autres entités telles que famille, interface ou configuration. Chacune des ces entités est associée à plusieurs objets (documentation, manuel...) dont l'évolution reste ignorée.

- La trace minimale et obligatoire insérée dans les listings est une trace locale, à très court terme. En effet, à l'issue de la lecture d'une réalisation, un utilisateur pourra trouver dans les deux premières lignes, le nom et l'auteur du catalogage précédent, le nom complet de l'objet et la liste de ses attributs. Il n'aura par conséquent aucune idée de la provenance ou de l'évolution globale de sa réalisation grâce à cette trace minimale, le journal n'étant qu'une trace facultative. Il ne pourra d'autre part connaître les raisons du catalogage ou des modifications apportées qu'en les demandant à l'intéressé, en espérant que ce dernier ne les ait pas oubliées.

C'est dans cette perspective que nous avons conçu et réalisé une seconde forme d'historique plus globale retraçant l'évolution des objets depuis leur naissance.
Cette deuxième forme consiste comme nous l'avons introduit au paragraphe 1.1.1. à adjoindre un nouveau type à chaque entité: le type HIST défini au paragraphe suivant.
Quant à la mise à jour globale de l'historique, elle va consister en la mise à jour des deux formes d'historique:

- l'insertion d'informations dans le texte des réalisa-

tions;
- la mise à jour de l'historique.

C'est ce second aspect de l'historique que nous allons maintenant examiner à travers ses deux fonctions essentielles, à savoir la mise à jour d'un historique et sa visualisation.

1.1.2.2.1. Définition du type historique.

Un historique est associé à une famille, une interface et une réalisation et son rôle essentiel est de pouvoir retracer l'évolution de ces divers objets du point de vue des modifications apportées aux différents types qui leur sont associés.

Définition.

Un type historique est une suite temporelle d'agrégats d'attributs où tout agrégat, associé à une modification, contiendra les informations suivantes:

(1) le nom de l'auteur de la modification ;
(2) la date de la modification :
(3) le type de l'objet modifié ;
(4) le numéro de la révision modifiée :
(5) le numéro de la révision créée :
(6) la source: le nom de la révision modifiée si celle-ci appartient à une version différente de celle de la révision créée :
(7) la destination: le nom de la révision créée si celle-ci appartient à une version différente de celle de la révision modifiée :
(8) un commentaire de l'utilisateur concernant sa modi-

fication.

Notons quelques particularités de ce type:

1) Remarquons d'abord, afin d'éviter toute ambiguité, que si l'utilisateur lit la révision A, la transforme et la recatalogue dans une nouvelle révision B, nous appelons A la révision modifiée et B la révision créée.

2) On remarque que certaines informations ((4), (5), (6) et (7)) ne sont utilisées que pour les réalisations. Une famille et une interface ne possèdent en effet pas de révisions concrètes et ne sont pas liées entre elles comme cela peut être le cas pour les réalisations.
Toute interface, famille ou réalisation possèdera la même structure d'historique, dans laquelle certains composants pourront être vides. Nous avons préféré ne pas introduire différents types d'historique pour des raisons d'homogénéité et de standardisation des opérations le concernant.

3) Les composants source et destination ((4) et (5)) sont exclusifs pour une même opération de modification.
Supposons en effet la situation suivante:

- on possède une version V1 existant en 3 révisions;
- on modifie V1.03 et catalogue la révision modifiée dans V2.01,

ce qui se traduit par le schéma suivant:

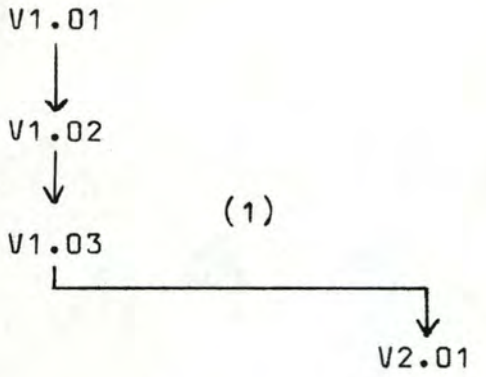


fig.1.2 Catalogage dans une nouvelle version.

- un historique associé à V1 et un autre à V2.

Il est tout à fait clair qu'une telle opération (1) va donner naissance à deux agrégats. Le premier relatera la modification de V1.03 et contiendra dans son champ destination, le nom de V2.01. Il fera partie de l'historique de V1. Le second dénotera la création de V2.01 et contiendra dans son champ source le nom de V1.03. Il fera partie de l'historique de V2. D'autre part, dans ce cas, le champ concernant le numéro de révision modifiée vaudra "03" pour V1 et "00" pour V2 et le numéro de révision créée vaudra "00" pour V1 et "01" pour V2.

4) Le champ commentaire représente la seule information qui n'est pas gérée automatiquement par le système. L'utilisateur sera invité à le composer et pourra y inclure tout ce qu'il désire, éventuellement rien, cette information ne répondant à aucune structure prédéfinie. Il pourra par exemple expliciter la raison de sa modification, les modifications apportées...

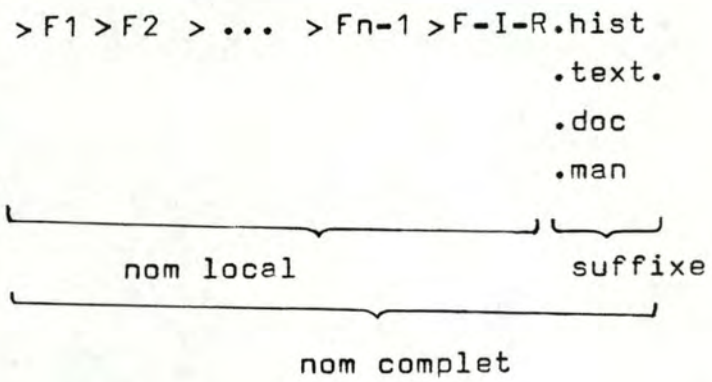
5) La valeur du champ date ((2)) d'un agrégat i est toujours inférieure ou égale à celle de l'agrégat i + 1 et les types ((3)) de deux agrégats consécutifs peuvent très bien être différents. Cela signifie

qu'un historique va relater la vie d'un objet d'une manière chronologique et que les modifications concernant un type particulier vont s'y répercuter à des endroits quelconques.

Quant à la désignation d'un historique, elle est semblable à celle de tout type associé, à savoir:

```
  >F1 >F2> ... >Fn-1 >Fn-I-R.hist   pour une réalisation,
  >F1 >F2 > ... >Fn-1 >Fn-I.hist    pour une interface,
et> F1 > F2 > ... > Fn-1 > Fn.hist   pour une famille.
```

Notons pour la suite de l'exposé ce que nous entendons par nom complet, nom local et suffixe dans la désignation d'un objet:



L'exemple pris pour une réalisation se généralise facilement au cas des interfaces et des familles.

Après avoir défini le type, nous allons maintenant expliciter les différentes opérations le concernant, à savoir sa mise à jour et sa visualisation.

1.1.2.2.2. Mise à jour d'un historique.

Rappelons (cfr 1.1.2.) qu'un historique est associé à toute famille, interface et réalisation. Son but est de retracer l'évolution de ces différents objets tant au point de vue de leur documentation ou de leur manuel que de leur texte ou de leur code binaire.
Lorsqu'une de ces entités (famille, interface, réalisation) est créée (cfr I;1.3.6.) par l'utilisateur, c'est-à-dire qu'elle se fait connaître de la base mais qu'aucun contenu ne lui est affecté, différents types associés (hist, doc, man, text) sont également créés et restent vides. L'historique d'une entité existante est par conséquent toujours présent dans la base. Il sera mis à jour lorsqu'un des types associés à l'entité sera modifié par un utilisateur et réintégré dans la base par la commande de catalogage. Remarquons qu'un historique ne peut en aucun cas être modifié par l'usager et que par conséquent, il n'existe pas d'historique du type hist des différentes entités concernées.

La mise à jour d'un historique revient à lui concaténer les informations explicitées en 1.1.2.2.1. (auteur, date, type de l'objet modifié, numéro de révision de l'objet modifié, numéro de révision de l'objet créé, source, destination, commentaire de l'utilisateur).

Rappelons toutefois que la création d'une nouvelle version à partir d'une version existante donne naissance à la mise à jour de deux historiques.

1.1.2.2.3. Visualisation d'historiques.

La mise à jour d'un historique ne sert pas à grand chose si l'utilisateur ne peut le visualiser. En outre, l'information présentée à celui-ci devra être une information pertinente, c'est-à-dire une information particulière de l'historique rejoignant la volonté de l'utilisateur, et non tout l'historique en lui-même. Cela semble en effet pour le moins être une tâche ardue que de retrouver une information précise dans une masse d'autres informations sur un listing ou défilant à l'écran.
Ainsi, pour permettre une plus grande classification et rapidité dans le choix de l'information par l'utilisateur du système, nous proposons d'établir des paramètres de visualisation comprenant chacun différentes options. De cette façon, toute personne désirant visualiser un historique pourra choisir une option parmi chaque paramètre et par conséquent obtenir l'information qu'elle désire.

Nous avons vu au paragraphe précédent que l'historique d'une version X était mis à jour chaque fois qu'une nouvelle révision de X était créée, ou qu'une version était créée à partir d'une révision de X, ou encore chaque fois qu'un manuel, une documentation ou un code binaire associé à X était modifié.
On peut ainsi facilement imaginer qu'après un certain temps, un historique devienne volumineux.
D'autre part, un utilisateur ne s'intéresse pas nécessairement à tout l'historique. Il peut seulement porter de l'intêret que pour une partie, par exemple de la semaine du 15-11-83 au 22-11-83. Un paramètre pour la date semble donc de rigueur.
Il est évident que la visualisation pourra s'opérer soit à l'écran, soit sur papier. Le mode de sortie constitue un nouveau paramètre.

Le résultat de la visualisation d'un historique est une suite d'agrégats de données où tout agrégat contient les informations suivantes: un nom, un type d'objet, un no de révision modifiée, un no de révision créée, le nom de la réalisation modifiée et celui de la réalisation créée, et un commentaire. Ce dernier a été fourni par un utilisateur de la base Adèle lors d'une session antérieure. Ce commentaire peut toutefois être assez long et simultanément gêner un utilisateur désirant jeter un rapide coup d'oeil sur l'historique. Un troisième paramètre permet dès lors de visualiser un historique avec ou sans les commentaires.

Rappelons (cfr.1.1.2.2.1) que le but d'un historique est de retracer l'évolution chronologique d'une famille, d'une réalisation, ou d'une interface à travers les différents objets qui leur sont associés (documentation, manuel, texte, codes binaires). Il peut être intéressant pour l'utilisateur de ne voir l'évolution que d'un seul de ces objets. Ceci constitue donc un nouveau paramètre qui permet de sélectionner à l'intérieur d'un historique, les agrégats de données concernant soit le manuel, soit la documentation, soit le texte, soit le code binaire, ou les quatre ensemble.

Pour les textes d'une réalisation, on peut constater pour un historique deux types d'agrégats:

- ceux qui relatent le fait de la création d'une nouvelle révision ;
- ceux qui relatent la création d'une nouvelle version.

D'une certaine manière, on peut dire que ces derniers sont plus importants et donc qu'il est plus intéressant que l'utilisateur puisse les obtenir facilement. Nous définissons ainsi le paramètre suivant qui permettra à l'usager de visualiser les agrégats importants de l'historique. On entend

par important, le premier agrégat, le dernier, et ceux donnant lieu à la création d'une nouvelle version.
Soit l'exemple suivant:

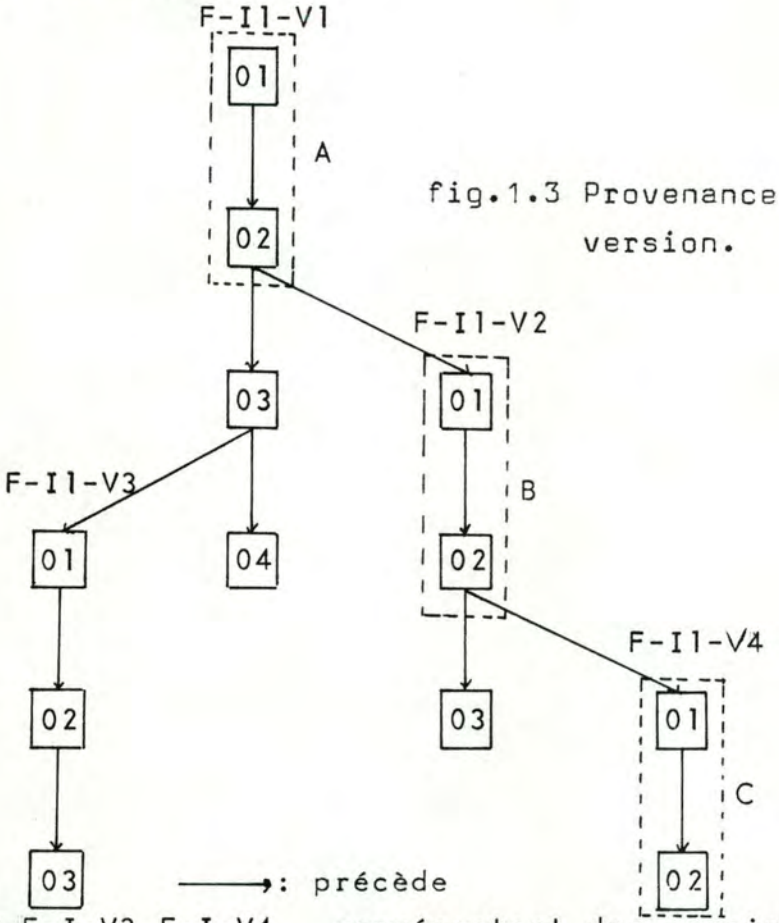


fig.1.3 Provenance d'une version.

F-IV1, F-I-V2, F-I-V3, F-I-V4 . représentent des versions

□ : représente un agrégat de données du type d'objet texte
: le chiffre à l'intérieur représente le numéro de révision

Il peut être intéressant pour l'utilisateur de connaître les agrégats de données correspondant à l'ensemble des révisions qui ont contribué à la création d'un version donnée. On définit cet ensemble comme étant la provenance de la version précisée. Si on applique cette définition à F-I1-V4, on obtient les ensembles A, B et C, tandis que pour une visualisation normale, le résultat serait l'ensemble C. Ceci constitue donc un paramètre complémentaire.
Tous les paramètres ci-dessus ont été choisi après discussions avec des utilisateurs de la base de programmes.

Tout objet se trouvant dans la base Adèle peut s'obtenir par la commande "lire" (cfr.I, 1.3.6.1.). Son format est

LIRE nombase [fichloc / ø]

Le paramètre nombase représente le nom d'une entité de la base et le paramètre fichloc désigne un fichier local de l'utilisateur. Le but de cette commande est de lire l'entité nombase en copiant son contenu dans fichloc si celui-ci existe, et à l'écran dans le cas contraire.

Par souci d'homogénéité avec le reste de la base, nous avons voulu que la visualisation d'un historique se fasse par la commande "lire". Nous avons donc été contraints de lui ajouter les paramètres décrits précédemment. Afin de ne pas obliger l'utilisateur à les mémoriser, nous leur donnons une valeur par défaut.
La commande "lire" pour le type historique est donc de la forme:

LIRE X.hist [fichloc / **ø**] [-doc / -man / -text / **-all**] [**-d** / -i / -d* / -i*] [d1,d2 / *,d2 / d1,* / **ø**] [-bf / **ø**]

—— : option par défaut

Définition des paramètres.

- X = nom de l'objet dans la base. Cet objet est soit une famille, une interface ou une réalisation.
- X.hist = nom de l'historique de cet objet.
- fichloc: nom du fichier local dans lequel on désire voir apparaître la visualisation. Par défaut, la visualisation sera effectuée à l'écran.

- type de l'objet demandé:
  - -doc: signifie que l'usager ne veut voir que l'historique de X.doc.
  - -man: idem pour X.man.
  - -text: idem pour X.text.
  - -all: option par défaut. Si c'est l'option demandée, l'historique de chaque composant de X est listé par ordre chronologique.

- type d'édition:
  nous avons regroupé ici les paramètres "importants" et "provenance".
  - -d: visualisation complète des éléments de l'historique demandé.
  - -i: signifie que l'usager ne désire visualiser que les éléments importants de l'historique. Rappelons qu'important signifie le premier, le dernier, et les éléments donnant lieu à une nouvelle version.
  - -{i,d}*: signifie que l'usager désire visualiser la provenance de la dernière révision de X.

- dates:
  quatre options sont possibles.
  - d1,d2: signifie qu'on visualisera les éléments de l'historique dont la date est comprise entre d1 et d2.
  - d1,*: idem pour [d1,→ [.
  - *,d2: idem pour] ←,d2] .
  - ∅: option par défaut. Elle signifie que tous les éléments de l'historique seront imprimés.

-bf: l'utilisateur se servira de cette option s'il désire une liste brève, c'est-à-dire sans commentaire dans les éléments listés.

Des exemples de visualisation seront donnés en annexe C.

1.1.3. Implémentation du type historique.

Comme nous l'avons vu en 1.1.2.2.1., le type historique est une suite d'agrégats. Chaque agrégat peut se subdiviser en deux parties: les 7 premiers éléments sont limités en longueur, et le dernier, le commentaire, est de longueur tout à fait quelconque.
Nous avons par conséquent décidé d'implémenter ce type comme un fichier de texte au sens général où Pascal l'entend: les enregistrements sont de taille variable et comportent chacun huit champs.
Afin d'éviter de perdre trop de place, les sept premiers champs sont implémentés en longueur variable. Chacun n'occupe que la place qui lui est strictement nécessaire et est terminé par un caractère NL (New Line). Tout champ vide ne comprend que le caractère NL.

Le fichier aura donc la structure suivante:

```
enregistrement 1
?*!
enregistrement 2
?*!
  :
  :
enregistrement n
?*!
```

où ?*! est un caractère spécial de délimitation d'enregistrement (ne pouvant apparaître dans les commentaires), et où tout enregistrement peut se résumer de la manière suivante:

nom du champ	longueur l	terminaison
auteur	$1 < l \leqslant 16$	NL
date	$l = 14$	NL
type (doc, man, text, co)	$2 \leqslant l \leqslant 4$	NL
no-révision 1	$0 \leqslant l \leqslant 3$	NL
no-révision 2	$0 \leqslant l \leqslant 3$	NL
source	$0 \leqslant l \leqslant 114$	NL
destination	$0 \leqslant l \leqslant 114$	NL
commentaire	indéterminé	NL

1.2. Conception globale du système et réalisation.

1.2.1. Architecture.

Sur le schéma présenté ci-dessous, nous mettons en évidence différents modules composant la gestion des historiques. La stratégie de découpe que nous avons suivie est surtout basée sur l'association d'une fonction à un module. Cette technique permet en effet de localiser facilement les erreurs et favorise le cas échéant la réutilisibilité des modules et l'extensibilité du système.
On peut remarquer que les deux modules fonctionnels (visualisation et mise à jour globale des historiques) s'intègrent dans le système de gestion de la base de la manière suivante:

- le module de visualisation est utilisé (au sens de la relation "utilise" définie dans la première partie en 1.3.2.1.) par le module de lecture de la base :
- le module de mise à jour globale des historiques est utilisé par le module de catalogage de la base.

Ces modules fonctionnels sont explicités en termes des différents composants qu'ils utilisent. La spécification de ces différents composants sera présentée au paragraphe suivant.

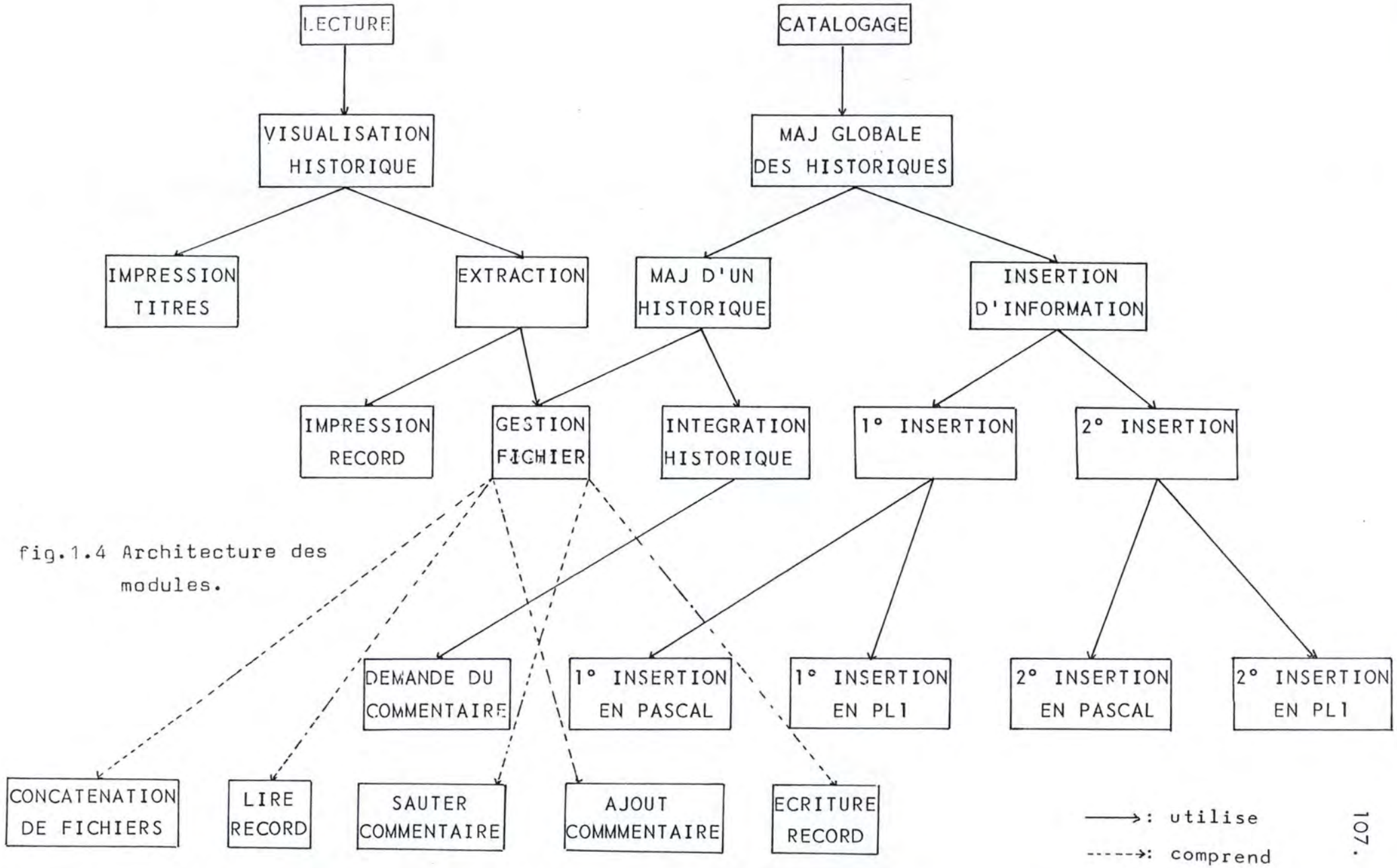


fig.1.4 Architecture des modules.

1.2.2. Mise à jour globale des historiques.(MAJGLHIST)

1.2.2.1. Spécification.

Ce module s'intègre en fait dans un cadre particulier que nous allons d'abord définir. L'utilisateur désire en effet cataloguer un objet dans la base. A cette fin, il lance d'une manière interactive la commande:

CATAL nombase fichloc (options).

Le paramètre nombase représente le nom d'une entité de la base et le paramètre fichloc désigne un fichier local de l'utilisateur. Le but du catalogage est de copier le contenu du fichier local dans l'entité concernée de la base (nombase). Toutefois, avant que cette copie ne soit réellement effectuée, la mise à jour globale des historiques est requise, comme nous l'avons vu plus haut.
Il est nécessaire à ce stade de mettre en évidence quelques particularités:

(1) Le contenu du fichier local est tout à fait quelconque. Un contrôle n'est effectué sur ce contenu que si l'objet nombase représente un manuel. Ce dernier ayant une structure bien définie, il convient de vérifier que le contenu du fichier local y répond. Dans tous les autres cas, aucun contrôle n'est préalablement effectué. La conséquence d'une telle situation est que l'utilisateur peut par inadvertance cataloguer un code objet dans le texte d'une réalisation. D'une manière plus générale, il peut involontairement cataloguer tout type d'objet dans un type qui lui est différent et qui est différent de manuel. Il peut d'autre part cataloguer le type X d'une entité Y dans le même type d'une entité Z (le texte d'une interface dans celui d'une réalisation...). Il est bien évident que ce type de possibilités ne peut que

nuire à l'utilisateur.
Pour éviter ce genre d'inconvénient, il faurdrait que l'utilisateur fournisse les deux noms d'objet dans sa commande de catalogage, mais la base n'a pas été conçue pour qu'il puisse utiliser cette façon de procéder.

(2) Si le contenu du fichier local est le texte d'une réalisation, il est important de remarquer qu'il peut être obtenu suivant deux filières:

- soit l'utilisateur l'a extrait de la base au moyen de la commande lire, l'a modifié pour ensuite le recataloguer;
- soit il a développé intégralement le texte dans sa propre directory pour ensuite le cataloguer.

Cette différenciation est importante puisque, dans le premier cas, le première ligne du texte possède une structure particulière. Elle contient en effet les informations suivantes:

%auteur : (1) %date : (2) %objet : (3) %

où (1) = nom de l'auteur du catalogage précédent
(2) = date du catalogage précédent
(3) = nom complet de l'objet précédemment catalogué.

Cela signifie qu'à l'issue des deux commandes

```
CATAL  >F> ... >Fn-I1-V1.text.02(*)  toto
LIRE   >F> ... >Fn-I1-V1.text.02   fich
```

la première ligne du fichier "fich" contient

```
%auteur : ... %date : ... %objet :
                          >F> ... >Fn-I1-V1.text.02 %
```

(*) Il est bien évident que ce paramètre peut être fourni d'une manière beaucoup plus condensée.

Ce fichier peut à son tour être modifié et catalogué par la commande

CATAL nombase fich

et à cet instant, nous possédons à la fois le nom de la destination (nombase) et celui de l'origine (>F> ... >Fn-I1-V1.text.02) que nous appellerons "nomlu".

(3) Si l'utilisateur désire commenter son catalogage, il doit utiliser l'option -com de la commande CATAL. A défaut de cette option, le commentaire n'est pas demandé.

La mise à jour globale des historiques va consister en la mise à jour du ou des historiques concernés par le catalogage et à l'insertion d'informations dans le texte des réalisations. Toutefois si le contenu du fichier local à cataloguer est le texte d'une réalisation, certains contrôles, stricts ou préventifs, doivent être effectués. Ceux-ci font partie du module MAJGLHIST.

a) Contrôle strict: ce type de contrôle arrête le traitement s'il n'est pas respecté. Dans le cas qui nous occupe, il s'agit d'empêcher que le texte d'une version V1 ne soit catalogué dans une version V2 si celle-ci existe et contient déjà un certain nombre de révisions.
L'opération (1) schématisée sur la figure suivante est par conséquent interdite:

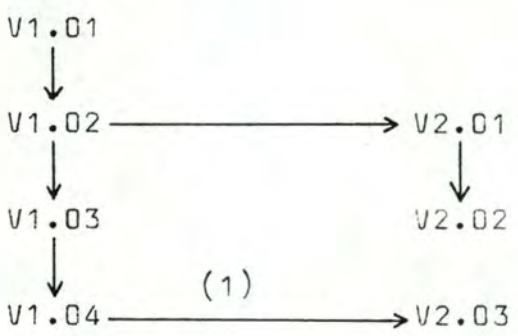


fig.1.5. Catalogage dans une version existante.

La commande "CATAL >F1> ... >Fn-I1-V2.text.03 fich" ne sera valide que si le nom complet d'objet trouvé dans la première ligne de fich possède le nom local >F1> ... >Fn-I1-V2.

b) Contrôle préventif: par ce type de contrôle, on prévient l'utilisateur dans certaines situations, et c'est celui-ci qui décide s'il faut continuer ou stopper le traitement. C'est ainsi que lorsque le nom d'objet n'est pas trouvé dans la première ligne du fichier local à cataloguer, l'utilisateur en est averti. Cela permettra d'éviter une partie des erreurs de catalogage dues à l'inattention. D'autre part, il se peut que le nom d'objet existe mais ne soit pas connu de la base. L'utilisateur peut en effet l'avoir altéré lorsqu'il a modifié son texte. Dans ce cas comme dans le précédent, il convient de prévenir l'utilisateur qu'il est impossible de déterminer ce qu'il catalogue et de lui demander confirmation. Il se peut finalement que le nom d'objet existe dans la première ligne du fichier local et qu'il contienne un nom local différent de celui contenu dans le paramètre nombase de la commande CATAL. Il convient dans ce dernier cas que l'utilisateur confirme sa décision de cataloguer son texte dans une nouvelle version.

A ces fins de contrôle, de mise à jour d'historiques et d'insertion d'informations, le module de mise à jour globale des historiques possède en entrée les arguments suivants:

- nombase: nom de l'objet dans lequel il faut cataloguer.
- fichloc: nom d'un fichier local contenant l'objet à cataloguer.
- comment: 1 si l'utilisateur désire commenter son catalogage.
  0 sinon.
- typenum: type numérique de l'objet à cataloguer.
- ptman: pointeur identifiant un manuel.

Les préconditions nécessaires à la bonne exécution du module sont les suivantes:

(1) Nombase doit être un nom complet. Il comprend donc le nom local d'une réalisation, interface ou famille, et un suffixe représentant le type de l'objet. Il représente un objet existant de la base.
Exemple:

>F> ... >Fn-I-V1.man

nom local — suffixe

(2) typenum = 5 si le type est doc
6 si le type est co
7 si le type est man
12 si le type est text (pour une interface)
13 si le type est text (pour un corps)
si nomlocal (:nombase) est le nom d'une interface, typenum ≠ 6 et 13
si nomlocal (:nombase) est le nom d'une famille, typenum ≠ 6, 12 et 13
si nomlocal (:nombase) est le nom d'une réalisation, typenum ≠ 12

Ces différentes contraintes sont imposées par l'environnement.

(3) Ptman n'existe que si l'objet à cataloguer est le texte d'une réalisation. Dans ce cas, il doit référencer le manuel associé à cette réalisation.

(4) L'historique nommé "nomlocal-hist" et le fichier local doivent exister.

(5) Le nom du fichier local doit avoir le format

nom.langage (Pascal, Pl1...)

Les effets du module peuvent se résumer de la manière suivante:

1) Le catalogage est invalide si l'objet à cataloguer est le texte d'une réalisation avec les conditions suivantes vérifiées:

- nomlocal(:nombase) ≠ nomlocal(:nombre)
et nomlu ≠ ' '
et no-révision(:nombase) ≠ '01'

où nomlu est le nom complet lu dans la première ligne du fichier local.

Dans ces conditions, un message d'erreur avertit l'utilisateur:

"Vous désirez cataloguer cet objet dans une nouvelle version."
nomlu → nombase
"catalogage refusé"

2) Le catalogage est arrêté par l'utilisateur. Cela se produit dans l'un des cas suivants, si l'objet à cataloguer est le texte d'une réalisation:

- nomlocal(:nombase) ≠ nomlocal(:nomlu)
et nomlu ≠ ' '
et no-révision(:nomlu) = '01'
et l'utilisateur désire arrêter :

- nomlu = ' ' ou nomlu est inconnu de la base ou il est d'un type différent de celui de nombase
et l'utilisateur désire arrêter ;

Dans l'une de ces conditions, un message avertit l'utilisateur:

"Votre objet n'est pas catalogué."
"Veuillez retaper la commande catal avec le nom correct de l'objet."

(3) Le catalogage est valide et désiré par l'utilisateur. Dans ces conditions, le module produit les résultats suivants:

- nomlu = ' ' si l'objet à cataloguer n'est pas le texte d'une réalisation, ou si le nom d'objet n'existe pas ou est inconnu dans la première ligne du fichier local;
  = nom d'objet contenu dans la première ligne du fichier local dans les cas contraires;

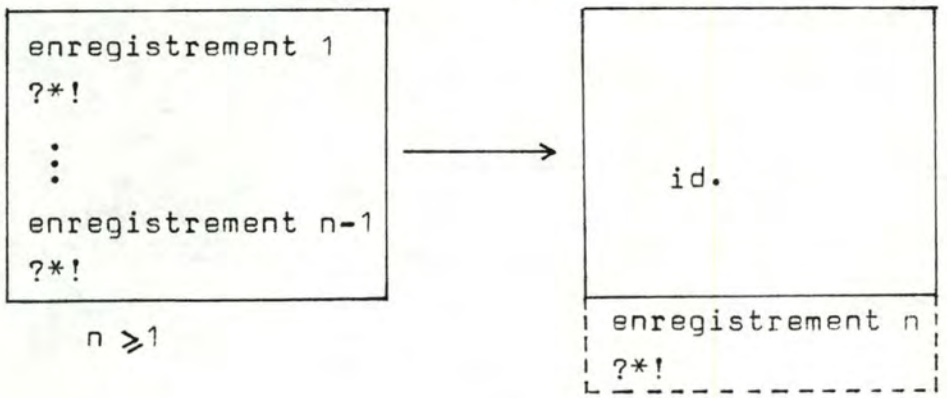
- nomlocal(:nombase).hist modifié;

- nomlocal(:nomlu).hist modifié si
  nomlocal(:nombase) ≠ nomlocal(:nomlu)
  et nomlu ≠ ' '

- fichloc modifié si l'objet à cataloguer est le texte d'une réalisation.

Ces résultats sont caractérisés par les postconditions suivantes:

(1) La modification de l'historique appelé nomlocal(:nombase).hist se schématise comme suit:



L'enregistrement n se caractérise de la manière suivante:

- auteur = Person-id.Project-id (ex.: DINSART.ADL);
- date = AA-MM-JJ..hhmn
  où AA = année
  MM = mois
  JJ = jour
  hhmn = heure et minute;
- les champs suivants dépendent de l'objet catalogué:
  - si nomlocal(:nombase) est le nom d'une famille
    - typeobjet = 'doc' si typenum = 5
      'man' si typenum = 7
    - les champs no-révision modifiée, no-révision créée, source et destination restent vides.
  - si nomlocal(:nombase) est le nom d'une interface
    - typeobjet = 'doc' si typenum = 5
      'man' si typenum = 7
      'text' si typenum = 12
    - les champs no-révision modifiée, no-révision créée, source et destination restent vides.
  - si nomlocal(:nombase) est le nom d'une réalisation
    - typeobjet = 'doc' si typenum = 5
      'co' si typenum = 6
      'man' si typenum = 7
      'text' si typenum = 13
    - si typeobjet ≠ 'text', les champs no-révision modifiée, no-révision créée, source et destination restent vides.
    - si typeobjet = text, ces champs prennent les valeurs suivantes:
      - no-révision modifiée = no-révision(:nomlu) si nomlu ≠ ' '
        = '00' si non
      - no-révision créée = no-révision(:nombase)

= '00' si nomlu ≠ ' '
et nomlocal(:nomlu)
≠ nomlocal(:nombase)

. source = vide
. destination = nomlu si
nomlocal(:nomlu) ≠
nomlocal(:nombase)
= vide si non.

- le champ commentaire correspond à une chaîne quelconque, éventuellement vide, de caractères n'incluant pas les sous-chaînes "NL?*!NL" et "NL.NL" où NL représente le caractère New Line.

(2) La modification de l'historique appelé nomlocal(:nomlu).hist se schématise de la même manière que celle qui vient d'être expliquée. Les champs auteur et date de l'enregistrement ajouté se caractérise de la même manière. Les autres sont quelque peu différents:

- typeobjet = 'text';
- no-révision modifiée = '00';
- no-révision créée = '01';
- source = nombase;
- destination = vide;
- commentaire = vide;

(3) La postcondition caractérisant la modification du fichier local sera explicitée en tant que postcondition du module d'insertion d'information. C'est en effet exclusivement cette insertion qui modifie le contenu du fichier local.

1.2.2.2. Construction de l'algorithme.

La structure de l'algorithme découle directement des résultats à produire. Il va en effet d'abord s'agir de contrôler le catalogage si l'objet à cataloguer est le texte d'une réalisation. Si le catalogage s'avère valide, il s'agira de mettre à jour les historiques concernés et d'insérer les informations dans l'objet à cataloguer si celui-ci représente le texte d'une réalisation.
L'algorithme va par conséquent s'orienter vers la structure suivante:

```
( préconditions )
( catalogage valide )
SI typenum = 13 ALORS
   Effectuer contrôles
   ( catalogage valide ou invalide )
SI (catalogage valide)  ALORS
   Mettre à jour historique
   SI typenum = 13  ALORS
      insérer informations
( postconditions)
```

Avant de raffiner les différents points, nous allons définir quelques outils existants de gestion de chaînes de caractères utilisés par ce module et par ceux qui seront explicités dans les paragraphes suivants:

- SUFFIXE:
  - entrée: chaîne(1,n);
  - sortie: préfixe, suffixe:
    - postcondition:
      - préfixe = chaîne(1,j) si
        chaîne(j+1) = '.' et 1 ⩽ j< n :
        = chaîne(1,n) si
        chaîne ne comprend ni '.' ni ' ' :
        = ' ' si
        chaîne(1) = '.' ou ' ' ;

```
        - postcondition (suite) :
            - suffixe = chaîne(i,n)  si
                  i > 1  et chaîne(i-1) = '.' ou ' ' ;
                       = ' '  si non ;

  - CONCATNOM:
      - entrée: chaîne1, chaîne2, délimiteur :
      - sortie: chaîne3 ;
        - postcondition:
            - chaîne3 = chaîne1délimiteurchaîne2 ;
```

Nous pouvons à présent raffiner le paragraphe des contrôles:

```
  nouvelle-version = faux ;
  EFFECTUER  Recherche (fichloc, nomlu) :
  SI nomlu ≠ ' '  ALORS
     VERIFNOM (nomlu, exist, type) ;
     SI (exist = faux)  OU  (type ≠ 13)  ALORS
        IMPRIMER  message-1 ;
        IMPRIMER  message-2 ;
        REPETER
            DEMANDE  réponse
        JUSQUE réponse = 'o' ou 'y' ou 'n' ;
        SI réponse ≠ 'n'  ALORS
           IMPRIMER  message-2;
           nomlu = ' ' ;
        SINON
           EFFECTUER fin ;
     SINON
        ( nomlu ≠ ' ', est connu de la base, et est du type
          textecorps )
        SUFFIXE (nomlu, nomlu-local, suffixe) :
        SUFFIXE (nombase, nombase-local, suffixe-base) :
        SI nomlu-local ≠ nombase-local  ALORS
           IMPRIMER  message-3 ;
           SUFFIXE (suffixe-base, préfixe, no-révision-base) :
           SI no-révision-base ≠ '01'  ALORS
```

```
                ( nomlocal(:nombase) ≠ nomlocal(:nomlu)  et
                  nomlu ≠ ' '  et
                  no-révision(:nombase) ≠ '01' )
                IMPRIMER  message-4 ;
                EFFECTUER fin ;
             SINON
                IMPRIMER  message-5 ;
                REPETER
                    DEMANDE réponse
                JUSQUE réponse = 'o' ou 'y' ou 'n' ;
                SI réponse = 'n'  ALORS
                   EFFECTUER fin
                SINON
                   nouvelle-version = vrai
  SINON
     ( nomlu = ' ' )
     IMPRIMER  message-6 ;
     IMPRIMER  message-5 ;
     REPETER
         DEMANDE réponse
     JUSQUE réponse = 'o' ou 'y' ou 'n' ;
     SI réponse = 'n'  ALORS
        EFFECTUER fin ;
```

où - message-1 = nomlu "n'est pas un corps ou est inconnu" :
- message-2 = nomlu "n'est pas pris en compte dans votre catalogage." ;
- message-3 = "Vous désirez cataloguer cet objet dans une nouvelle version."
  nomlu ----→ nombase
- message-4 = "catalogage refusé" ;
- message-5 = "Confirmez-vous votre catalogage (o-n) ?" ;
- message-6 = "Attention! Il est impossible de savoir ce que vous cataloguez." ;
- fin est une procédure imprimant le message
  "Votre objet n'est pas cataloguer."
  "Veuillez retaper la commande catal avec le nom correct de l'objet."

et stoppant le traitement.

- VERIFNOM est une routine existante vérifiant l'existence d'un objet passé en paramètre, et renvoyant son type ;
- Recherche est une procédure se définissant comme suit:
  - entrée: fichloc: nom du fichier local ;
  - sortie: nomlu ;
    - postcondition:
      - nomlu = ' ' si
        la première ligne de fichloc n'inclut pas la chaîne %objet : ... %
        = nom si
        elle inclut la chaîne %objet : nom %

A l'issue de ces contrôles, nous sommes dans l'un des états suivants:

- catalogage refusé ;
- catalogage accepté et
  (nomlu ≠ ' ' et nomlocal(:nomlu) ≠ nomlocal(:nombase))
  ssi
  no-révision(:nombase) = '01' et nouvelle-version = vrai ;

En suivant les différentes postconditions du module et en étant assuré que le catalogage est valide, nous pouvons raffiner la seconde partie de l'algorithme:

```
( initialisations de la mise à jour )
DATEHEURE (date) ;
NOMPERSONNE (auteur) ;
SI typenum = 5   ALORS  typeobjet = 'doc'
             6                    'co'
             7                    'man'
             12,13                'text'
( mise à jour )
SI typenum = 13  ALORS
   SI nouvelle-version = vrai  ALORS
      ( màj de deux historiques )
      no-rev-créé = '00' ;
      SUFFIXE (nomlu, nomlu-local, suffixe) ;
```

```
        SUFFIXE (suffixe, préfixe, no-rev-modifié) ;
        source = ' ' ;
        destination = nomlu ;
        MAJHIST (auteur, date, typeobjet, no-rev-modifié,
                 no-rev-créé, source, destination, nomlu-local,
                 comment ) ;
        no-rev-créé = '01' ;
        no-rev-modifié = '00' ;
        source = nombase ;
        destination = ' ' ;
        SUFFIXE (nombase, nombase-local, suffixe) ;
        MAJHIST (auteur, date, typeobjet, no-rev-modifié,
                 no-rev-créé, source, destination,
                 nombase-local, 0) ;
     SINON
        ( nouvelle-version = faux )
        SUFFIXE (nombase, nombase-local, suffixe) ;
        SUFFIXE (suffixe, préfixe, no-rev-créé) ;
        SI nomlu ≠ ' '  ALORS
           SUFFIXE (nomlu, nomlu-local, suffixe) ;
           SUFFIXE (suffixe, préfixe, no-rev-modifié)
        SINON
           no-rev-modifié = '00' ;
        source = ' ' ;
        destination = ' ' ;
        MAJHIST (auteur, date, typeobjet, no-rev-modifié,
                 no-rev-créé, source, destination,
                 nombase-local, comment) ;
     INSINF (nombase, fichloc, date, auteur, ptman) ;
  SINON
     ( typenum ≠ 13 )
     no-rev-créé=no-rev-modifié=source=destination=' ' ;
     SUFFIXE (nombase,nombase-local,suufixe) ;
     MAJHIST (auteur, date, typeobjet, no-rev-modifié,
              no-rev-créé, source, destination,
              nombase-local, comment) ;
( postconditions )
```

où - DATEHEURE et NOMPERSONNE sont des routines existantes qui renvoient respectivement la date sous forme AA-MM-JJ..hhmn, et l'auteur sous la forme Person-id.Project-id ;
- INSINF et MAJHIST sont deux modules explicités aux paragraphes 1.2.3. et 1.2.4. .

1.2.3. Insertion d'information dans le texte des réalisations. (INSINF)

1.2.3.1. Spécification.

Ce module est utilisé par la mise à jour globale des historiques. Sa fonction consiste à insérer différentes informations (cfr. 1.1.3.1.) dans le texte des réalisations. A cette fin, il reçoit en entrée les arguments suivants:

- nombase : nom complet de l'objet dans lequel il faut cataloguer.
- fichloc : nom du fichier local contenant le corps à cataloguer.
- date : date du jour sous le format AA-MM-JJ..hhmn.
- auteur : responsable du catalogage sous le format Person-id.Project-id.
- ptman : pointeur identifiant un manuel.

Les préconditions nécessaires au bon déroulement du module sont les suivantes:

1) Le type de l'objet à cataloguer doit être le texte d'une réalisation. Nombase a par conséquent un format du type: >F> ... >Fn-I-V.text.ij

2) Le pointeur ptman doit référencer le manuel de nomlocal(:nombase) (ex.: >F> ... >Fn-I-V)

3) Le fichier local doit exister et avoir un nom de format: nom.langage.

Le seul résultat produit résulte dans la modification du fichier local. Celle-ci se caractérise par les postconditions

suivantes:

1) Toute occurence des chaînes %auteur % ou %auteur : ... % est remplacée par
%auteur : person-id.Project-id %.

2) Toute occurence des chaînes %date % ou %date : ... % est remplacée par
%date : AA-MM-JJ..hhmn %.

3) Toute occurence des chaînes %objet % ou %objet : ... % est transformée en %objet : nombase %.

4) Toute occurence de la chaîne %journal % provoque l'insertion d'une ligne entre la ligne contenant cette chaîne et la ligne suivante. La ligne insérée possède le format suivant:

```
$ Person-id.Project-id - AA-MM-JJ..hhmn - nombase $
```

où le signe $ représente les délimiteurs de commentaire associé à un langage particulier.

```
$ ... $ représente  (* ... *) en Pascal
                    /* ... */ en Pl1
```

etc.

5) Toute occurence de la chaîne %attribut % ou %attribut : ... % est remplacée par %attribut : ligne attributs %. La ligne d'attributs a le format suivant: attr = val ; attr = val; ... ; attr = val. Sa longueur est inférieure ou égale à 60 caractères. Les couples attributs-valeurs sont ceux du manuel associé à nombase et qui se nomme nomlocal(:nombase).man. S'il n'existe aucun attribut pour la réalisation nomlocal(:nombase), la ligne d'attributs a pour valeur 'aucun'.

6) Si la liste des attributs est telle qu'elle ne peut s'insérer intégralement dans une ligne attributs, un nombre quelconque de lignes sont insérées entre la ligne contenant la chaîne %attribut % ou %attribut : ... % et la ligne suivante. Ces lignes ont le format suivant:

$ %attribut (suite) : ligne attribut % $

Tous les attributs du manuel sont ainsi pris en compte.

7) Les premières lignes de fichloc ont toujours la structure suivante:

ligne 1 $ %auteur : auteur %date : date %objet : nombase % $
ligne 2 $ %attribut : ligne attributs % $
ligne 3 $ %attribut(suite) : ligne attributs % $
⋮
ligne n $ %attribut(suite) : ligne attributs % $

avec $n \geqslant 2$.

1.2.3.2. Alternatives de réalisation.

Ce module s'exécutera pendant la commande de catalogage (cfr. 1.2.2.). Cette commande étant interactive, il est par conséquent impératif que l'insertion d'informations s'effectue en un minimum de temps. Nous avions le choix de l'implémenter intégralement en Pascal SOL ou d'utiliser des macros d'édition. Nous disposions en effet de l'éditeur de texte TED qui possédait un langage de programmation primaire à partir duquel on pouvait construire des macros.
L'avantage de tout implémenter en Pascal est bien évidemment la portabilité. Mais nous avons constaté lors de tests qu'un traitement consistant à lire un fichier d'entrée pour le recopier sur un fichier de sortie prenait plus ou moins huit fois plus de temps en Pascal que via l'éditeur Ted. C'est la raison qui nous a poussés à utiliser Ted pour effectuer les différentes subsitutions. Insistons donc sur le fait que nous avons privilégié l'efficacité à la portabilité. Ceci a surtout été motivé par le fait que nous travaillions dans un contexte d'expérimentation et de recherche et que la base n'était nullement destinée à être transférée sur des systèmes différents.
Le module d'insertion d'informations va donc utiliser des macros pour réaliser les transformations définies dans ses postconditions. Remarquons que pour pouvoir utiliser des macros à partir d'un programme Pascal, il est nécessaire de passer par l'intermédiaire de commandes appelées exec-com. Ces dernières peuvent comporter toutes les commandes du système et quelques instructions de contrôle comme le if ou le goto. C'est à partir de ces exec-com que les macros Ted seront invoquées. Le type d'implémentation que nous avons adopté se schématise donc de la manière suivante:

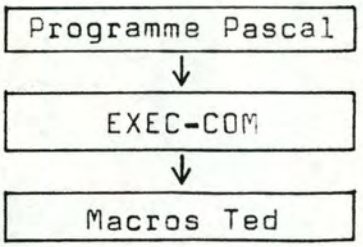


fig.1.6 Enchaînement.

⟶ : précède

1.2.3.3. Définition des exec-com utilisées.

1.2.3.3.1. Première insertion. (PREINS.EC)

Ce module se charge d'effectuer les substitutions d'auteur, de date, d'objet, du journal et d'une ligne d'attributs.
A cette fin, elle reçoit en entrée les paramètres suivants:

- l'auteur (Person-id.Project-id) ;
- la date (AA-MM-JJ..hhmn) ;
- nombase ;
- ligne attributs (attr=val; attr=val; ... : attr=val) ;
- fichloc : nom du fichier où il faut substituer ;
- langage : langage dans lequel est écrite la réalisation contenue dans fichloc.

Certaines conditions doivent être vérifiées avant son exécution:

1) Aucun des paramètres ne peut contenir le caractère " ", Ceci résulte du fait qu'une exec-com est appelée à partir d'un programme Pascal en fournissant au Command-Processor une ligne de format: ec nom de l'exec-com (paramètres).
Les différents paramètres sont séparés par un ou plusieurs caractères " ".

2) La longueur de la ligne d'attributs est inférieure ou égale à 60 caractères.

3) le fichier fichloc doit exister.

A l'issue du traitement, le fichier fichloc est modifié et les postconditions (1), (2), (3), (4), (5), (7) du module d'insertion d'informations sont vérifiées.
Pour réaliser sa fonction, cette exec-com utilise les macros "Macropascal.Ted" si le langage = Pascal et MacroPl1.Ted si le langage = Pl1. Nous avons par conséquent opté pour une politique de redondance maximum au niveau des macros. Celles-ci sont en effet rédigées dans un langage possédant peu de structure (affectation, instructions if et goto) et incluent une série de commandes d'édition qui ne sont pas toujours des plus évidentes. Nous avons par conséquent décidé de réduire leur complexité au maximum en vue d'une maintenance aisée.
Par exemple, si un nouveau langage est souhaité (ex.: Cobol), il suffira d'ajouter une macro (ex.: macrocobol.Ted) qui ne différenciera des deux autres qu'à travers les délimiteurs de commentaires (ex.: * en COBOL).
Ces différentes macros reçoivent toutes en entrée la date, l'auteur, le nom complet de l'objet (nombase) et une ligne d'attributs. Leur exécution résulte en la modification du fichier local caractérisée par les postconditions (1), (2), (3), (4), (5) et (7) du module d'insertion d'informations. Elles peuvent se résumer de la manière suivante:

- SI la ligne 3 de fichloc contient la chaîne %attribut(suite)
  ALORS SUPPRIMER toutes les lignes contenant %attribut(suite) ;
- SUPRRIMER les n premières lignes de fichloc commençant par
  $ %a (n 0) ;
- INSERER en ligne 1 et 2, les lignes:
  $ %auteur%date%objet% $
  $ %attribut% $
- SUBSTITUER toutes les chaînes %auteur(n caractères qcques)% ($n \geq 0$)
  PAR %auteur : auteur % :
  (postcondition (1) vérifiée)

- SUBSTITUER toutes les chaînes %date (n caract.)% (n 0)
  PAR %date : date % ;
  (postconditions (1) et (2) vérifiées)
- SUBSTITUER toutes les chaînes %objet (n caract.)% (n 0)
  PAR %objet : nombase % :
  (postconditions (1), (2) et (3) vérifiées)
- SUBSTITUER toutes les chaînes %attribut (n caract.)% (n 0)
  PAR %attribut : ligne d'attributs % :
  (postconditions (1), (2), (3), (5) et (7) vérifiées)
- INSERER après toute ligne contenant %journal% une ligne de format $ auteur - date - objet $:
  (postconditions (1), (2), (3), (4), (5) et (7) vérifiées)

Le caractère $ symbolise le délimiteur de commentaire:

$ ... $ (* ... *) en Pascal
/* ... */ en Pl1
* ... en Cobol.

Une autre exec-com se charge en partie de la postcondition (6) du module d'insertion d'informations. Nous poursuivons par sa définition.

1.2.3.3.2. Seconde Insertion. (ATTRSUIV.EC)

Ce module se charge de l'insertion des lignes d'attributs excessifs. Il reçoit en entrée:

- une ligne d'attributs (ligneattr) :
- le nom du fichier local (fichloc) ;
- le langage .

Les préconditions sont identiques à celles de la première insertion et comptent en plus la condition suivante:

(1) Le fichier local doit au moins contenir une ligne incluant la chaîne %attribut : ... %.

En sortie,le fichier local est modifié. La modification est

caractérisée par la postcondition suivante:

(1) Toute ligne possédant la chaîne *attribut : ... %
est suivie par la ligne $ %attribut(suite) : ligneattr % $
où les signes $ symbolisent des délimiteurs de commentaire.
De la même manière qu'à la première insertion, cette exec-com utilise les macros attrsuivpl1.Ted et attrsuivpascal.Ted suivant que le langage utilisé est Pl1 ou Pascal.

1.2.3.4. Construction de l'algorithme d'insertion.

(1) La structure générale de l'algorithme va par conséquent refléter les deux insertions successives:

(préconditions de INSINF)
(1) EFFECTUER première insertion :
(2) TANT QU'il existe des attributs
EFFECTUER seconde insertion :
(postconditions de INSINF)

(2) Avant d'exécuter (1), les préconditions de la première insertion doivent être respectées:

- les préconditions de INSINF entraînent que:
  - date, auteur, nombase, fichloc, langage ne comprennent aucun ' ' ;
  - le fichier local existe :
- il reste alors à satisfaire:
  - ligne d'attributs ne comprend aucun ' ' :
  - longueur(ligne d'attributs) $\leqslant$ 60 :
- ce qui conduit pour la première insertion à:
  EFFECTUER mise en forme attributs :
  EFFECTUER première insertion ;

où la routine de mise en forme attributs se définit

- entrée: ptattr: pointeur sur les attributs :
  - précondition:
    (1) ptattr doit référencer au moins un attribut
    (ptattr ≠ nil)

- sorties: ligneattr: ligne d'attributs :
  ptattr: pointeur courant sur les attributs :
  - postconditions:
    (1) ligneattr ne comprend aucun ' ' ;
    (2) longueur(ligneattr) ⩽ 60 :
    (3) ligneattr de format: attr=val: ... ;attr=val:
    (4) si ptattr ≠ nil alors
    longueur(ligneattr) + longueur(ptattr(attr);
    ptattr(val)) > 60 :

(3) Avant d'exécuter (2), les mêmes conditions doivent être respectées. En outre, la précondition de ATTRSUIV.EC doit être vérifiée.

(4) Ces considérations ont conduit à l'algorithme suivant:

```
( préconditions de INSINF )                                    (1)
ptattr = ptman(pointeur attribut) ;
SUFFIXE (fichloc, préfixe, langage) ;
SI ptattr ≠ nil  ALORS
   ( ptattr ≠ nil entraîne précondition de mise en forme
     attributs )
   EFFECTUER mise en forme attributs (ptattr, ligneattr) :
   ( postconditions de mise en forme attributs )               (2)
SINON  ligneattr = 'aucun' ;                                   (3)
( ((1) et (2)) ou ((1) et (3)) entraîne la précondition
  de PREINS.EC )
EFFECTUER  PREINS.EC (auteur, date, nombase, fichloc,
                      langage, ligneattr) ;
( postcondition (1), (2), (3), (4), (5) et (7) de INSINF )     (4)
```

TANT QUE ptattr ≠ nil
    (ptattr ≠ nil entraîne la précondition de mise en forme attributs)
    EFFECTUER mise en forme attributs (ptattr, ligneattr) :
    (postcondition de mise en forme attributs) (5)
    ((4) et (5) entraîne la précondition de ATTRSUIV.EC)
    EFFECTUER ATTRSUIV.EC (ligneattr, fichloc, langage) :
    (postcondition de ATTRSUIV.EC)
(U postconditions de ATTRSUIV.EC entraîne la postcondition (6) de INSINF) (6)
((4) et (6) entraîne postconditions de INSINF)

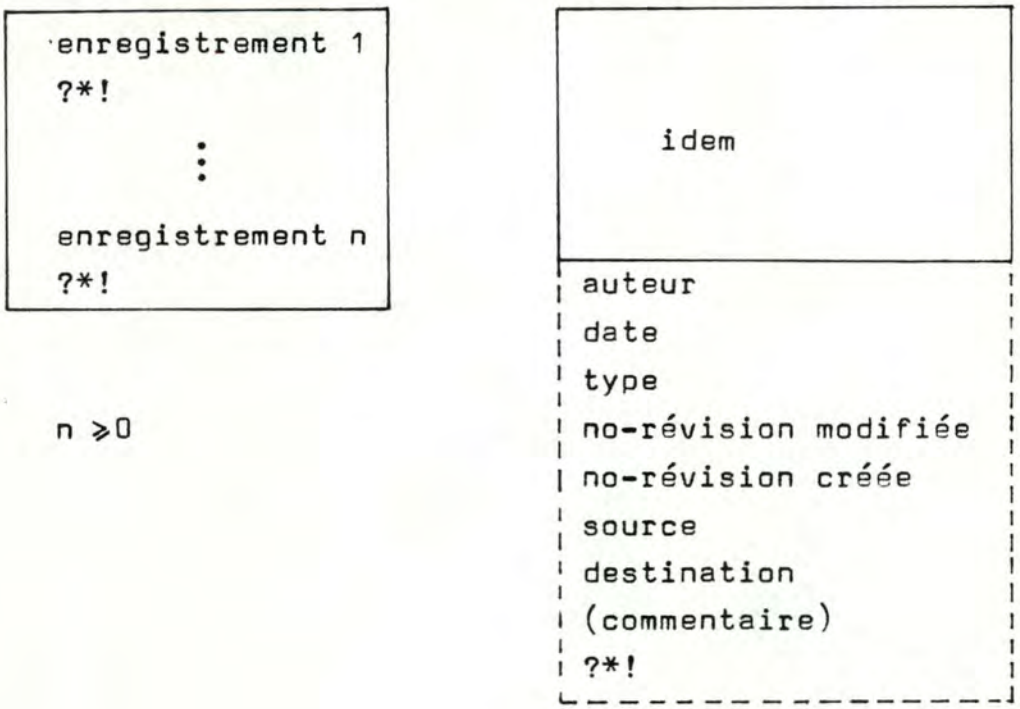
1.2.4. Mise à jour d'un historique. (MAJHIST)

1.2.4.1. Spécification.

Ce module a pour but d'étendre un historique en y adjoignant un nouvel agrégat contenant ses huit attributs (cfr.1.1). Il reçoit par conséquent les sept informations habituelles (auteur, date, type, no-révision modifiée, no-révision créée, source et destination) le nom local de l'objet modifié, et un indicateur valant 1 si le commentaire est désiré et 0 sinon.
Il est supposé qu'à l'entrée de ce module, les différentes informations sont cohérentes. Leur cohérence n'est pas vérifiée et par conséquent, le module sous-entend les préconditions suivantes:

1) les nos de révisions, la source et la destination sont vides si l'objet catalogué n'est pas le texte d'une réalisation;
2) le type d'objet a une valeur compatible avec le nom local de l'objet modifié (le type text est par exemple incompatible avec une famille);
3) les informations date et auteur sont fournies sous leur bon format (AA-MM-JJ..hhmn pour date et Person-id.Project-id pour auteur);
4) les informations source et destination n'ont pas simultanément une valeur;
5) le fichier "nomlocal.hist" doit exister et peut être vide.

En sortie, le fichier de nom "nomlocal.hist" est modifié de la manière symbolisée ci-après:



1.2.4.2. Choix de réalisation.

Tout comme l'insertion d'informations, ce module s'intègre dans l'exécution de la commande interactive de catalogage. Il est dès lors impératif qu'il prenne le moins de temps possible. C'est la raison qui nous a poussés à utiliser des exec-com et des macros Ted dont l'efficacité n'était plus à prouver. Celles-ci vont servir à demander le commentaire à l'utilisateur et à concaténer les informations à l'historique concerné. Il est évident que de telles opérations peuvent être lourdes en Pascal alors qu'elles deviennent simples et efficaces sous un éditeur. C'est ainsi que la politique que nous avons adoptée se résume de la manière suivante:

- créer en Pascal un fichier de travail appelé !Brecord qui contient les sept premières informations d'un élément d'historique;
- utiliser des macros et des exec-com qui vont concaténer le fichier temporaire (!Brecord) au fichier historique, y ajouter un commentaire fourni par l'utilisateur et clôturer l'ensemble par le délimiteur ?*!.

Afin de montrer avec quelle aisance une concaténation de deux fichiers s'opère sous Ted, considérons les commandes ci-après:

il suffit simplement d'effectuer les opérations suivantes:

```
Ted          (invoquer l'éditeur)
bx           (se placer dans le buffer x)
r file1      (lire sur le fichier file1 dans x)
r file2      (lire sur le fichier file2 dans x)
w file1      (recopier le contenu de x sur le
              fichier file1)
q            (quitter l'éditeur)
```

1.2.4.3. Définition de l'exec-com utilisée.

(CONSTHIST.EC)

Ce module possède en entrée les informations suivantes:

- nom-hist : nom du fichier historique à mettre à jour:
- comment : paramètre valant 1 si le commentaire est désiré
  0 sinon;
- fichier !Brecord.

Il est nécessaire qu'avant son exécution, les conditions suivantes soient respectées:

1) le fichier historique concerné doit exister et peut être vide;
2) le fichier !Brecord ne peut être vide.

A l'issue du traitement, le fichier !Brecord est détruit et le fichier historique concerné est modifié de la manière décrite en 1.2.4.1. lors de la spécification de MAJHIST.
Ce module utilise la macro COMMENT-Vide.Ted si le commentaire est désiré et le fichier historique vide, et COMMENT.Ted si le commentaire est désiré et si le fichier historique con-

tient déjà un certain nombre d'enregistrements.

1.2.4.3.1. Définition des macros utilisées.

Les deux macros utilisées sont en réalité presqu'identiques. La raison d'une telle duplication provient du fait que lorsque le commentaire est demandé à l'utilisateur, il est directement concaténé aux fichiers historique et !Brecord, et ce dans un but de rapidité. Cela signifie que la macro va devoir:

- lire le fichier historique dans un buffer x;
- lire !Brecord dans le même buffer x;
- demander le commentaire qui viendra s'ajouter à x;
- recopier x sur le fichier historique.

Et comme la lecture d'un fichier vide sous Ted provoque une erreur, nous avons dédoublé la macro.
Les deux macros reçoivent en entrée le nom de l'historique à modifier.
COMMENT-VIDE.Ted exige que le fichier historique soit vide et COMMENT.Ted exige qu'il ne le soit pas. Les deux exigent que le fichier !Brecord existe et ne soit pas vide.
Elles produisent en sortie le même résultat, à savoir la modification de l'historique d'une manière conforme à celle décrite en 1.2.4.1.
Elles présentent cependant la postcondition suivante:

(1) Le commentaire est une chaîne de caractères quelconque (éventuellement vide) qui n'inclut pas les sous-chaînes "NL?*!" et "NL.NL", NL représentant le caractère New Line.

COMMENT.Ted peut par exemple se résumer de la manière suivante:

```
    LIRE fichier historique dans le buffer x :
    LIRE !Brecord dans x ;
(3) DEMANDER une ligne de commentaire dans x :
(4)     SI ligne entrée = "." ALORS
                REMPLACER ligne entrée PAR "?*!" ;
                ECRIRE    x sur le fichier historique
        SINON
(5)             SI ligne entrée = "?*!" ALORS
                   SUPPRIMER ligne entrée ;
                   IMPRIMER  "ligne incorrecte" ;
                ALLER en (3) ;
```

Les lignes (4) et (5) permettent de vérifier la postcondition (1).

1.2.4.3.2. Construction de CONSTHIST.EC.

La construction de cette exec-com est en fait très simple. Elle relève des deux conditions d'entrée:

- commentaire demandé (d) ou non ($\bar{d}$);
- fichier historique vide (v) ou non ($\bar{v}$);

ce qui conduit à quatre cas différents repris dans l'algorithme suivant:

```
( préconditions (1) et (2) de CONSTHIST.EC )
SI v̄.d  ALORS
   ( préconditions (1) et (2) et v̄ entraîne précondition
     de COMMENT.TED )
   APPEL de COMMENT.TED ;
   ( postcondition de COMMENT.TED )
   DETRUIRE !Brecord ;
   ( postcondition de COMMENT.TED et !Brecord détruit
     entraîne postcondition de CONSTHIST.EC )
(fin SI)
```

```
SI v.d  ALORS
   ( préconditions (1) et (2) et v entraîne précondition
     de COMMENT-VIDE.TED )
   APPEL de COMMENT-VIDE.TED ;
   DETRUIRE !Brecord ;
   ( postcondition de COMMENT-VIDE.TED et !Brecord détruit
     entraîne postcondition de CONSTHIST.EC )
(fin SI)

SI v̄.d  ALORS
   LIRE fichier historique dans le buffer x ;
   LIRE !Brecord dans x ;
   ECRIRE ?*! dans x ;
   ECRIRE le buffer x sur le fichier historique ;
   DETRUIRE !Brecord ;
   ( postcondition de CONSTHIST.EC )
(fin SI)

SI v̄.d̄  ALORS
   LIRE !Brecord dans le buffer x ;
   ECRIRE ?*! dans x ;
   ECRIRE le buffer x sur le fichier historique ;
   DETRUIRE !Brecord ;
   ( postcondition de CONSTHIST.EC )
(fin SI)
```

1.2.4.4. Construction de l'algorithme de mise à jour.

Grâce à ces macros et exec-com, l'algorithme se réduit à sa plus simple expression. Il suffit en effet de créer un fichier temporaire puis d'invoquer CONSTHIST.EC qui se charge d'effectuer la mise à jour.
Il a par conséquent la structure suivante:

```
( précondition de MAJHIST )                                    (1)
OUVRIR !Brecord ;
EFFECTUER  ECRITURE-RECORD ( auteur, date, type, no-révi-
                             sion modifiée, no-révision
                             créée, source, destination,
                             !Brecord ) :                      (2)
FERMER !Brecord ;
CONCATNOM ( nomlocal, 'hist', '.', nom-fich-hist ) :
( (1) et (2) entraîne précondition de CONSTHIST.EC )
EFFECTUER CONSTHIST.EC ( nom-fich-hist, indicateur de
                         commentaire ) ;
( postcondition de CONSTHIST.EC ssi
  postcondition de MAJHIST )
```

1.2.5. Visualisation d'historique. (VISHIST)

1.2.5.1. Spécification.

Comme nous l'avons vu au paragraphe 1.1.2.2.3. de cette partie, la commande d'activation de ce module se présente sous la forme suivante:

LIRE X.hist [fichloc / ∅] [-doc / -man / -text / -all] [-d / -i / -d* / -i*] [d1,d2 / d1,* / *,d2 / ∅] [-bf / ∅]

—: option par défaut

En fait, ce module s'intègre dans le cadre de la commande générale de la base "lire". Cette commande constitue un module à part entière appelé "lire". Sa fonction est de valider la commande soumise par l'utilisateur, et ensuite d'aiguiller le traitement vers le module adéquat. Cela signifie que si une commande est correcte et que l'objet à lire est du type historique, le module lire appellera le module vishist.
Celui-ci recevra en entrées:

(1) Le nom de l'historique que l'on veut visualiser.
(2) Le nom du fichier de sortie sur lequel on mettra le résultat.
(3) Le type d'objet qui prend l'une des cinq valeurs suivantes: doc, man, co, text, all.
(4) Le type d'édition qui prend l'une des quatre valeurs suivantes: d, d*, i, i*.
(5) Deux dates.
(6) Brève: prend les valeurs 0 ou 1.

Il faut comme préconditions:

(1) Le fichier des historiques doit exister et être

ordonné par date croissante.

(2) Chaque entrée doit avoir une valeur définie.

Le résultat de ce module se trouvera sur le fichier de sortie. Celui-ci comprendra un titre caractérisant les options des paramètres de la commande lire, suivi d'un ou plusieurs ensembles d'agrégats de données du type historique. Un ensemble étant associé à un historique, le terme "plusieurs" caractérise le fait que l'utilisateur a demandé une "provenance" (cfr.1.3.2.2.). Tous les agrégats sont ordonnés par ordre croissant de dates et répondent aux options des paramètres de la commande lire.

1.2.5.2. Procédé de résolution.

Le résultat de la commande LIRE Vk.hist est,par exemple:

HISTORIQUE de Vk

Vk.man	créé le 10-11-83	par CASTAIGNE.ADL
Vk.doc	créé le 10-11-83	par CASTAIGNE.ADL
Vk.text.01	créé le 10-11-83	par CASTAIGNE.ADL
Vk.text.01	modifié le 10-11-83	par DINSART.ADL

On peut remarquer qu'il est composé d'un titre suivi d'un ensemble d'agrégats de données associé à Vk.hist.
Supposons que Vk est issue de la version V1, elle-même issue de la seconde révision de Vm. Si nous demandons la provenance de Vk, c'est-à-dire que nous soumettons la commande "lire Vk.hist -d*", le résultat sera, par exemple:

PROVENANCE DE Vk

```
Vm.man              créé le 10-10-83          par DINSART.ADL
Vm.text.01          créé le 10-10-83          par DINSART.ADL
Vm.text.01          modifié le 11-10-83       par DINSART.ADL
Vm.text.02          modifié le 12-10-83       par DINSART.ADL
   création de Vl.text.01

Vl.man              créé le 12-10-83          par DINSART.ADL
Vl.text.01          créé le 12-10-83          par DINSART.ADL
A PARTIR DE Vm.text.02
Vl.text.01          modifié le 13-10-83       par CASTAIGNE.ADL
Vl.text.02          modifié le 20-10-83       par CASTAIGNE.ADL
Vl.text.03          modifié le 10-11-83       par CASTAIGNE.ADL
   création de Vk.text.01

Vk.man              créé le 10-11-83          par CASTAIGNE.ADL
Vk.doc              créé le 10-11-83          par CASTAIGNE.ADL
Vk.text.01          créé le 10-11-83          par CASTAIGNE.ADL
A PARTIR DE Vk.text.03
Vk.text.01          modifié le 10-11-83       par DINSART.ADL
```

Ce résultat est composé d'un titre suivi de trois ensembles d'agrégats de données des types historiques associés respectivement à Vm, Vl et Vk. En fait, au départ on ne connaît que Vk. Pour connaître Vl, il faudra le chercher dans Vk.hist, et Vm pourra être trouvé dans Vl.hist.

On voit donc se dessiner la structure de l'algorithme:

- extraire du fichier historique (de Vk), les agrégats de données qui répondent aux options des paramètres de la commande. Ensuite, nous les formatons pour les placer sur un fichier temporaire. Du fichier historique, nous retenons également le nom de la version qui a donné naissance à Vk (Vl).

tout ceci est réalisé par le module EXTRACTION que nous définirons au paragraphe 1.2.5.4..

- nous sauvons le fichier temporaire sur le fichier de sortie dont le nom est passé en paramètre à ce module. Ensuite nous recommençons le processus d'extraction sur la nouvelle version trouvée (V1). Nous concaténons alors le fichier temporaire et le fichier de sortie. Nous réitérons ce processus jusqu'à ce qu'il n'y ait plus de version origine d'une autre.

- ensuite le module IMPRESSION-TITRES (cfr.1.2.5.3.) fournit un fichier contenant un titre caractérisant les options de la commande lire soumise par l'utilisateur. Nous concaténons le fichier de sortie à ce fichier et nous obtenons ainsi le fichier escompté.

Il faut remarquer qu'une visualisation normale (cfr.1er exemple) est un cas particulier de cet algorithme. En effet, ce cas n'occasionne qu'une seule itération.

1.2.5.3. Impression des titres.

Pour une visualisation d'historique, il existe plusieurs types d'éditions possibles. Chacun est caractérisé par les paramètres de la commande lire. Il est nécessaire que l'utilisateur puisse distinguer les différentes éditions. Il est dès lors utile d'insérer en tête de liste un titre synthétisant les paramètres de la commande.
En entrée, nous aurons les paramètres de la commande, à savoir:

- type d'édition qui prend comme valeur d,d*,i ou i* :
- type d'objet qui prend comme valeur man,doc,text ou co :

- date-inf qui est la première date, éventuellement égale à "*" ;
- date-sup qui est la seconde date, éventuellement égale à "*" ;
- nom d'objet: l'objet dont on demande la visualisation de l'historique.

En entrée, nous aurons aussi le nom d'un fichier vide. Le résultat sera placé dans ce fichier.

1.2.5.4. Extraction.

1.2.5.4.1. Spécification.

Ce module est appelé par VISHIST (cfr.1.2.5.2.). Son but premier est d'extraire d'un fichier historique, dont le nom est passé comme paramètre, les enregistrements répondant aux options des paramètres de la commande lire soumise par l'utilisateur, et de les formater sur un fichier de sortie dont le nom est également fourni en paramètre.
Rappelons que dans le cas d'une visualisation "provenance", VISHIST réalise les extractions sur les historiques une à une. Ceci ne peut se faire que si après chaque extraction, on lui fournit le nom de la version suivante à traiter.
Ce nom se trouve dans l'historique que le système est en train de traiter. Cela constitue le second but d'extraction qui est donc de trouver le nom de la version qui a donné naissance à la version qui est en cours d'extraction.
Etant donné que ce petit traitement ne se fait pas à chaque visualisation, nous avons ajouté un paramètre supplémentaire au module extraction: le paramètre option-source qui permet de signaler si oui ou non il faut réaliser le traitement.
Toujours concernant la provenance, il est à remarquer que la procédure d'extraction se termine soit à la fin, soit au milieu du fichier historique. Pour mieux comprendre ce fait, prenons un exemple:

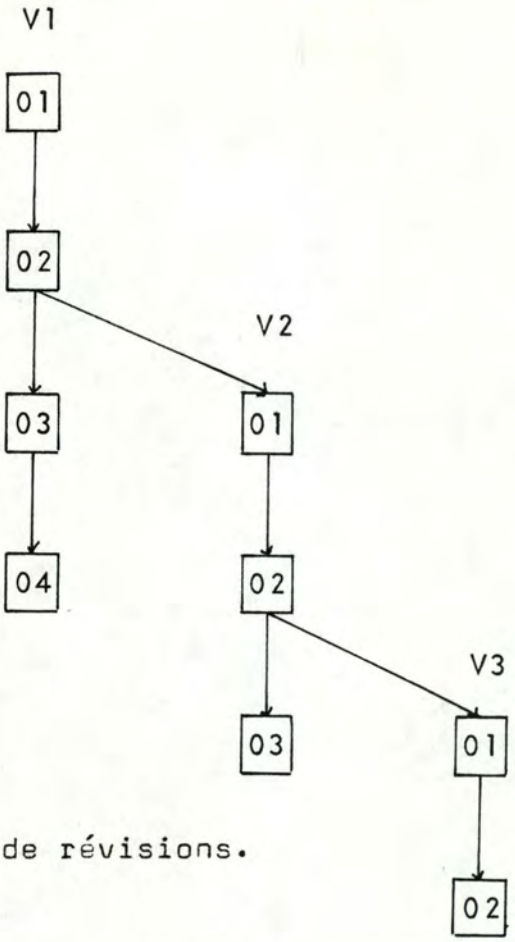


fig.1.7 Arboresence de révisions.

V1,V2 et V3 représentent trois versions ayant respectivement 4,3 et 2 révisions. V3 a été créée à partir de la seconde révision de V2 qui a elle-même été créée à partir de la seconde révision de V1.

Si on demande la provenance de V3:

> la procédure d'extraction se termine pour V3 à la fin du fichier historique et pour V1 et V2 après leur seconde révision.

Pour réaliser cela, nous avons ajouté un paramètre destination qui comprendra le nom de la version qui a été créée à partir de la version entrain d'être traitée. Si on traite V2, destination comprendra V3. Il en est de même entre V1 et V2. Par contre, si on traite V3, ce traitement n'est pas à faire. C'est pour cette raison que nous ajoutons un paramètre option-destination qui permettra de dire s'il faut ou non réaliser ce traitement.

En entrée, nous aurons donc:

- le nom d'un fichier historique :
- le nom d'un fichier de sortie :

- type d'objet qui prend une des valeurs man, doc, text, all ;
- deux dates ;
- important prenant la valeur 0 ou 1 ;
- option-source ;
- option-destination ;
- destination comprenant le nom de la version qui a été créée à partir de la version qu'on traite ;
- brève prenant la valeur 0 ou 1.

Il faut comme précondition que l'historique soit ordonné par date croissante.

Les résultats sont:

- un fichier de sortie qui comprend une sous-suite des éléments d'historiques formatés du fichier historique de départ (voir postconditions) ;
- une variable source qui comprend le nom de la version qui a donné naissance à la version dont on est en train de traiter l'historique.

Les postconditions sont les suivantes:

- si le type d'objet est égal à doc,man,co ou text, le fichier comprendra uniquement des éléments d'historique relatifs respectivement à la documentation, au manuel, au code objet et au texte. S'il est égal à all, le fichier comprendra les éléments d'historique relatifs à tous les types d'objets (man, doc, text, co) ;
- la date de tous les éléments repris sera comprise dans l'intervalle de dates.
- important = 1 et le fichier de sortie comprend tous les premiers et derniers éléments d'historique de chaque type d'objet ainsi que tous ceux qui relatent le fait de la création d'une nouvelle version.

- si option-destination = 1, les éléments d'historique repris dans le fichier de sortie sont tous ceux antérieurs à celui qui relate la création de la version dont le nom se trouve dans destination.

Les sept premiers champs des enregistrements du fichier historique repris dans le fichier de sortie seront formatés comme indiqué dans la spécification de impression-record. Le commentaire sera présent uniquement si brève égale 0. Finalement, si option source égale 1, la variable source comprendra le nom de version qui a créé la version dont on est en train de traiter l'historique.

1.2.5.4.2. Impression-record.

1.2.5.4.2.1. Spécification.

Cette procédure est appelée par le module extraction. Elle lui sert à formater sur un fichier, dont le nom est passé en paramètre, un agrégat de données du type historique.

En entrée, il y aura:

- un agrégat de données comprenant:
  - un nom d'auteur de la modification ;
  - une date de modification ;
  - le type de l'objet modifié ;
  - le numéro de la révision modifiée ;
  - le numéro de la révision créée ;
  - source ;
  - destination ;
- le nom de la version dont on extrait les agrégats de données de l'historique :
- le nom d'un fichier de sortie qui peut soit être vide, soit comprendre des agrégats de données déjà formatés du type historique.

Le résultat sera le fichier de départ auquel on a ajouté l'agrégat de données dans un format agréable (nous n'entrons pas ici dans les détails de formatage).

1.2.6. Spécification de la gestion du fichier.

A la base de la gestion des historiques se retrouve le type historique. Au paragraphe 1.1.4., nous avons vu qu'il est composé d'une suite d'enregistrements séparés par un délimiteur (?*!). Chaque enregistrement est composé de huit champs se terminant tous par NL (New Line). Les sept premiers sont caractérisés par le fait que leur longueur maximale est connue tandis que le huitième, le commentaire, est de longueur inconnue. Une seconde caractéristique de ce dernier est qu'il est directement éditable, c'est-à-dire qu'il peut être affiché à l'écran ou imprimé sur papier sans devoir subir un traitement préalable. Ceci contrairement aux sept autres champs auxquels il faudra insérer des blancs, ajouter des titres pour pouvoir les éditer.
Voici les différentes opérations possibles sur le fichier des historiques:

- LIRE-RECORD: lit les sept premiers champs d'un enregistrement d'historique dont le nom est passé en paramètre.
- ECRITURE-RECORD: imprime les sept premiers champs d'un enregistrement d'historique à partir de la position courante dans un fichier dont le nom est passé en paramètre.
- AJOUT-COMMENTAIRE: imprime un commentaire,débutant à la position courante d'un fichier, sur un autre fichier à partir de sa position courante. Les deux noms de fichiers sont passés en paramètre.

- SAUTER-COMMENTAIRE: saute un commentaire sur un fichier dont le nom est passé en paramètre. Cette procédure est utilisée lors de l'option brève de la commande lire.
- CONCAT: cette procédure reçoit trois noms de fichiers en entrée. Le résultat se trouvera sur le troisième fichier et comprendra la concaténation des deux premiers.

2. Un système de mémorisation de versions.

2.1. Première spécification.

2.1.1. Objectifs.

Dans tout système à révisions multiples, il s'avère nécessaire de garder au moins deux révisions. Il n'est en effet pas rare que des erreurs de manipulation conduisent à la destruction d'une révision et il est dès lors primordial de prévoir une "roue de secours". C'est la raison pour laquelle la base ADELE permettait à ses utilisateurs de conserver les deux dernières révisions de chaque version. Mais dans le but de permettre à l'usager de reconsidérer tous ses choix antérieurs, cette conservation minimale devient insuffisante. Il est dès lors nécessaire de garder toutes les révisions.
Il est cependant évident qu'une telle conservation devient rapidement très coûteuse pour de gros logiciels contenant des centaines de révisions. Elle résulte, en effet, en un gaspillage de place mémoire d'autant plus inacceptable que les différences entre deux révisions successives sont généralement minimes. Il convient par conséquent de réaliser cette conservation intégrale en économisant un maximum de place. Ces deux objectifs nous ont conduits à développer un nouveau système de mémorisation des révisions. Il est évidemment nécessaire qu'il soit totalement transparent à l'utilisateur et qu'il soit suffisamment efficace afin de ne pas nuire à l'intérêt de l'utilisation de la base de programmes.

D'autre part, il est fréquent, dans un système où différentes versions sont liées entre elles, qu'une erreur existant dans une version se soit automatiquement répercutée vers ses filles. Il est dans ce cas intéressant de pouvoir corriger la version mère et de reproduire automatiquement la correction sur toutes les filles.
Supposons en effet qu'on ait introduit, par inadvertance, une

erreur dans une version A. Tant que personne ne s'en aperçoit, cette erreur demeure dans toutes les versions descendantes de A. On pourrait concevoir un outil qui, après la correction par l'utilisateur de la version A, répercuterait la modification dans ses versions descendantes.
Cela permet naturellement d'épargner un temps considérable. Dans cette optique, nous avons pourvu la base d'un outil de fusion de deux versions.

2.1.2. Concepts utilisés.

La différence entre deux révisions successives du point de vue de leur contenu est souvent minime. En effet, elle ne comprend que les quelques modifications que l'utilisateur a apportées à une révision pour créer la suivante.
Au lieu de garder toutes les révisions entièrement, il est donc avantageux de ne garder que cette différence.
Nous avons repris la notion de delta telle qu'elle est définie dans RCS (cfr.première partie, 2.4.) où l'unité de changement est la ligne. Si un caractère change dans la ligne, nous considérons que toute la ligne est modifiée.
Pour rappel, un delta entre un texte A et un texte B est constitué de l'ensemble de lignesincluses dans A et pas dans B et vice versa.
Un delta (i,j) permet de reconstituer le texte de la révision i à partir du texte de la révision j.
Notre objectif est de régénérer une révision à partir d'une révision gardée en clair et d'un delta.
La meilleure façon de modifier un texte étant d'utiliser un éditeur de texte, nous avons pensé, tout comme RCS, à inclure dans les deltas des commandes d'édition. Un delta devient alors un ensemble de commandes d'un éditeur applicables directement sur le texte d'une révision donnée. Ceci facilite grandement la tâche de reconstitution du texte d'une révision mais est à mettre au passif de la portabilité.

2.1.3. Alternatives de conception.

Dans ce paragraphe, une étude personnelle sur la mémorisation des révisions de versions multiples sera présentée. Nous nous intéressons plus particulièrement dans une première partie au cas d'une version unique. Dans un second temps, nous reprendrons les considérations de la première partie que nous appliquerons aux arbres de versions.

2.1.3.1. Cas d'une version unique.

Dans ce qui précède, nous avons parlé d'une révision gardée en clair. Le problème est de déterminer laquelle.
Supposons d'abord qu'il n'y ait qu'une seule version. Nous avons donc une suite de révisions que nous représentons à la figure suivante où ri ⟶ ri+1 indique que la révision i+1 a été créée à partir de la révision i:

fig.2.1 Version unique.

Une première possibilité est de garder la première révision en clair (ce que nous schématisons en fig.2.2 en représentant le noeud r1 par un carré), et de déterminer un delta(j,1) direct à la création de toute révision j (j >1).

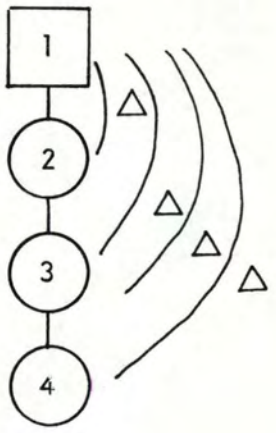


fig.2.2 Première possibilité.

La reconstitution de rj (j≠1) implique donc simplement l'application du delta(j,1) direct sur r1. La lecture de r1 est directe tandis que la lecture de rj (j≠1) nécessite sa reconstitution.
Le catalogage d'une nouvelle révision rj (j≠1) demande le calcul d'un nouveau delta(j,1).

Une autre possibilité est de garder la première révision en clair et de déterminer un delta(j,j-1) à la création de toute rj (j≠1). La figure suivante résume cette possibilité:

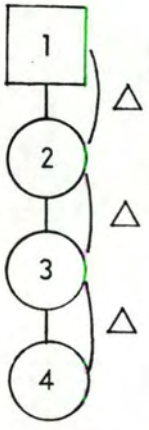


fig.2.3 Seconde possibilité.

La reconstitution de rj implique donc la reconstitution régressive de tous les rk à partir de r1 et des deltas

(rk,rk-1) directs correspondants (k=j,j-1, ... ,2).
La lecture de r1 est directe et celle de rj (j≠1) demande sa reconstitution régressive.
Le catalogage de rj implique le calcul du delta(j,j-1) direct et par conséquent la reconstitution de rj-1.
Il est à remarquer qu'un delta entre deux révisions successives est moins volumineux qu'un delta calculé entre deux révisions quelconques. Un avantage de la seconde possibilité par rapport à la première serait donc un gain de place mémoire.

Une troisième possibilité est de garder la dernière révision ri en clair et de déterminer le delta(i-1,i) inverse à sa création. La figure ci-dessous résume cette possibilité:

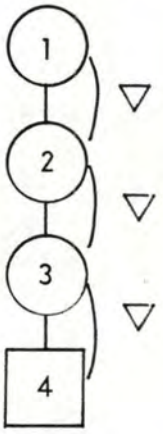


fig.2.4 Troisième possibilité.

La reconstitution d'une rj implique la reconstitution régressive de toutes les rl à partir de la dernière révision ri et des deltas(l-1,l) inverses correspondants (l=i,i-1, ... ,j+2,j+1).
La lecture de la dernière révision est directe tandis que la lecture de la révision j implique sa reconstitution régressive.
Le catalogage d'une révision j (j≠1) implique chaque fois le calcul du delta(j-1,j) inverse.

Une quatrième possibilité a été directement représentée sur la figure suivante:

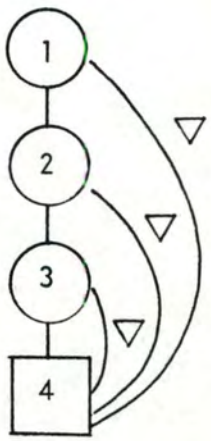


fig.2.5 Quatrième possibilité.

La reconstitution d'une rj implique donc simplement l'application du delta(j,i) inverse sur la dernière révision ri.
La lecture de la dernière révision est directe et la lecture de rj nécessite sa reconstitution (j≠no dernière révision).
Pour le catalogage d'une révision j (j≠1), j-1 deltas doivent être recalculés. En effet, si nous regardons sur la figure suivante, nous remarquons que pour cataloguer la révision 5, nous devons calculer delta (1,5), delta (2,5), delta (3,5) et delta (4,5).

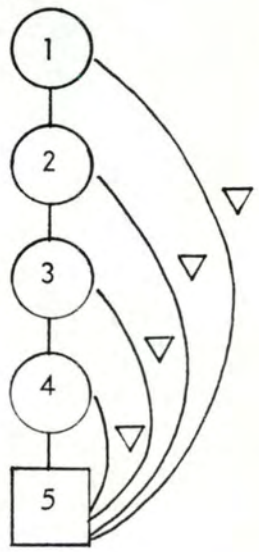


fig.2.6 Catalogage d'une nouvelle révision.

Le choix d'une des quatre solutions n'est pas évident. Il dépend évidemment des opérations que l'on fera sur la base. La lecture est quasiment toujours faite sur la dernière révision. Il est donc coûteux de devoir sans cesse régénérer cette révision à la lecture comme cela est rendu obligatoire dans les deux premières solutions.
Le catalogage dans la dernière solution est très coûteux puisqu'il impose le calcul de j-1 deltas pour pouvoir mémoriser la révision j.
C'est pour ces raisons que nous avons choisi la troisième solution qui consiste à garder la dernière révision en clair et à calculer les deltas sur des révisions consécutives, ce qui les rend moins volumineux.

2.1.3.2. Cas d'un arbre de versions.

Si nous appliquons la solution du paragraphe précédent sur l'arbre des versions, nous obtenons la figure suivante:

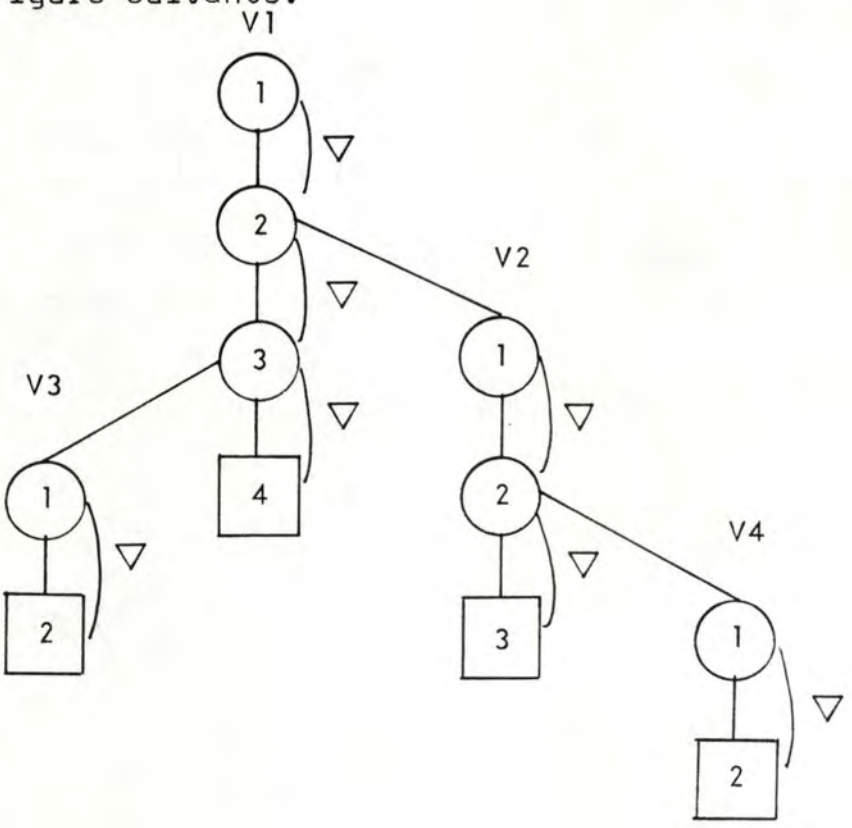


fig.2.7 Application à un arbre de versions.

On remarque cependant que les versions ont souvent beaucoup en commun, ce qui suggère que nous pouvons également gagner de l'espace disque en n'en stockant qu'une seule intégralement.
Toujours sur base de la discussion du paragraphe précédent, l'idée serait de garder la dernière révision de la dernière version créée. Les révisions de cette dernière seront mémorisées sous forme de deltas inverses tandis que les autres versions seront composées de deltas inverses puis de deltas directs comme le montre la figure suivante, où on suppose que V4 est la dernière version créée:

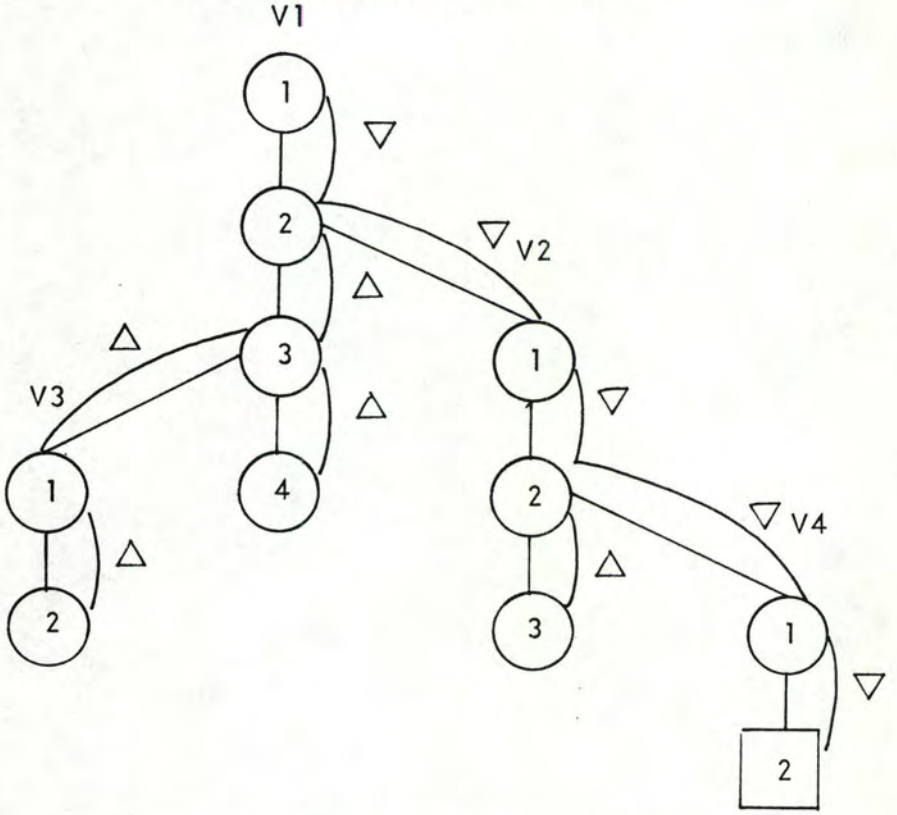


fig.2.8 Première possibilité.

Si maintenant nous créons la version 5 à partir de V1.4, nous remarquons sur cette figure :

- que tous les deltas inverses de V2 et V4 sont convertis en deltas directs :

- que les deltas directs(3,2) et (4,3) de V1 deviennent des deltas inverses(2,3) et (3,4).

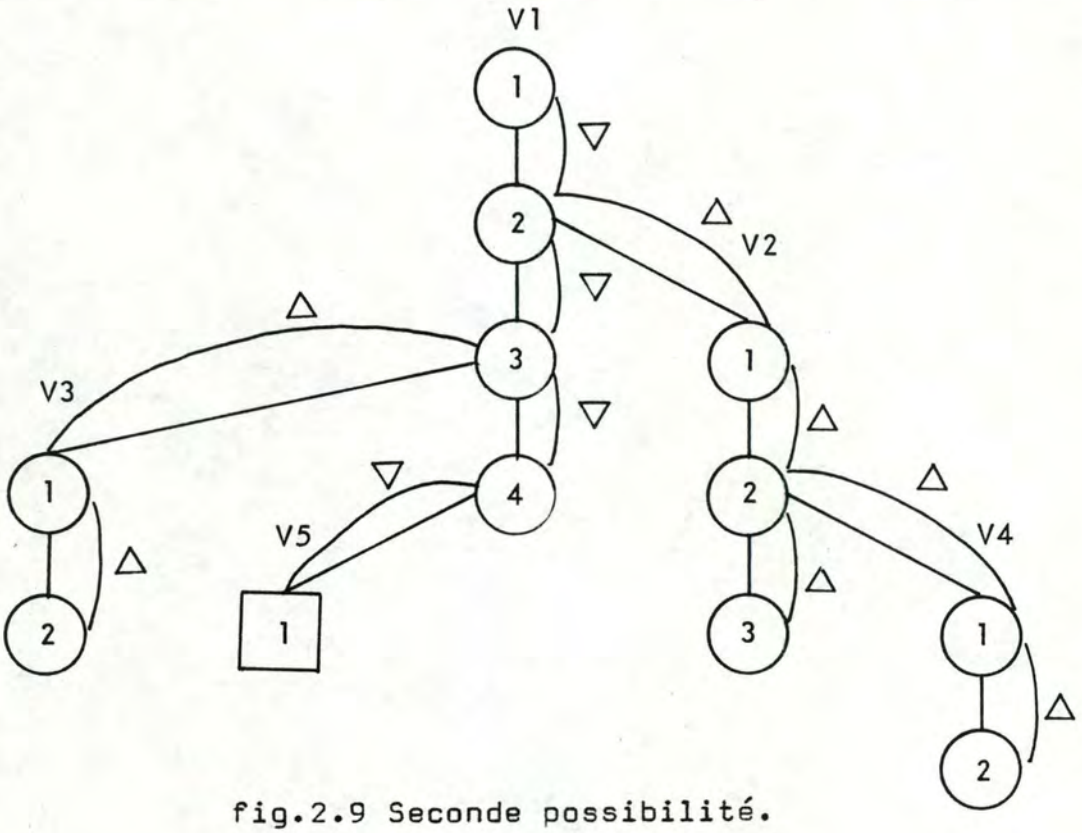


fig.2.9 Seconde possibilité.

On se rend compte du coût d'une telle solution lors de la création de certaines nouvelles versions. De plus, il peut y avoir un mélange de deltas directs et inverses au sein d'une même version (cas de V1 et V2). C'est pourquoi nous nous orientons vers un autre type de solution.
Nous choisirons une version, dans l'arbre de versions, qui sera celle dont la dernière révision sera gardée en clair. Une telle version sera appelée version tronc.
Les versions branches seront celles dont aucune révision n'est gardée en clair.
Nous obtenons ainsi la figure suivante:

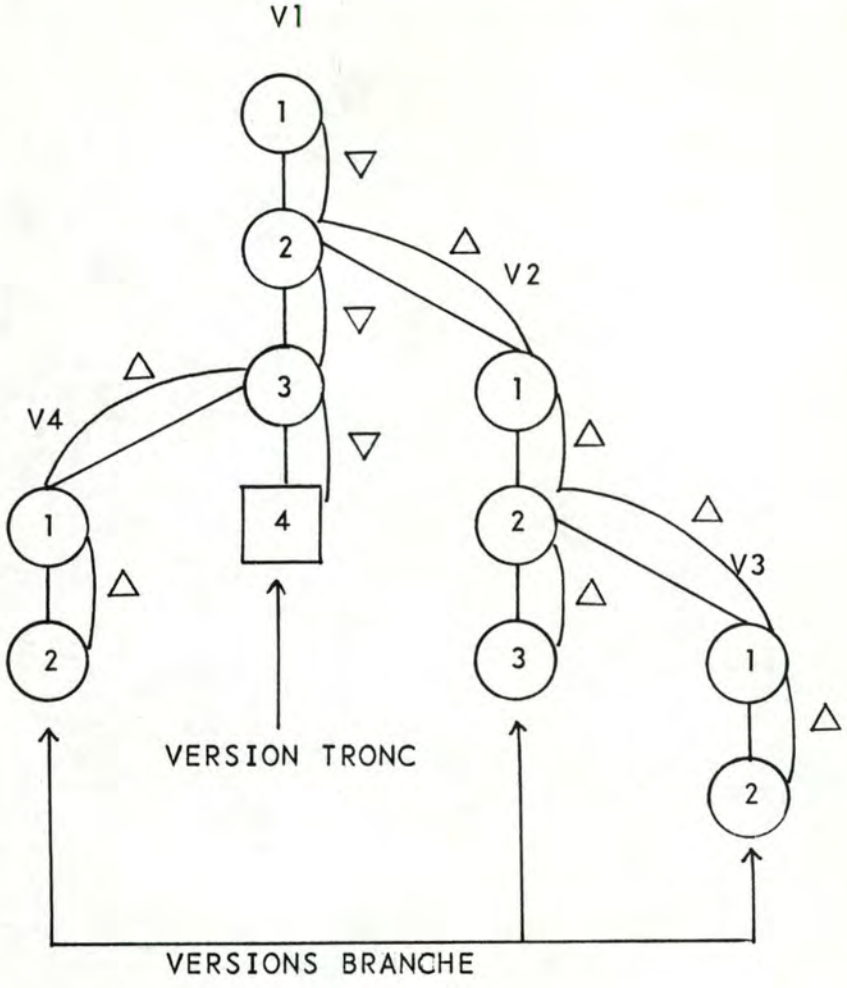


fig.2.10 Versions branches et version tronc.

Dans ce cadre, nous introduisons le concept d'ordre de version (ord(V)). Il vaut 0 pour une version tronc, et pour une version branche, il vaut le nombre de versions qui ont été créées à partir du tronc pour l'obtenir. Ainsi, sur la figure 2.8,on a ord(V1)=0, ord(V2)=ord(V3)=1, et ord(V4)=2.
Nous allons dans ce qui suit évaluer la solution suggérée à la figure2.10 du point de vue de la reconstitution, lecture et catalogage d'une révision d'une version branche. Pour les révisions du tronc, la discussion est identique à celle présentée à la troisième possibilité du paragraphe précédent.

La reconstitution d'une révision dans une version branche d'ordre k issue de la révision R d'une version tronc, requiert l'utilisation d'au moins k deltas. Après la recons-

titution de la révision R de la version tronc (cfr. §précedent), il faut reconstituer successivement la première révision de chaque version branche d'ordre inférieur à k. Ceci est réalisé à partir des deltas directs (1,R), (2,1), (3,2), ..., (k-1,k-2), (k,k-1). On obtient ainsi la première révision de la version branche d'ordre k.

La lecture d'une révision d'une version branche demande à chaque fois sa reconstitution d'une façon identique à celle que nous venons d'expliquer.

Le catalogage d'une révision dans une version branche demande toujours la reconstitution d'une révision, soit la révision précédente quand on catalogue au sein d'une même version, soit la révision de la version dont elle est issue lorsqu'on catalogue dans une nouvelle version branche. Une fois cette révision reconstituée, on peut calculer le delta direct entre cette dernière et la nouvelle.

De cette discussion, il découle que le coût de reconstitution d'une révision est, dans cette solution, proportionnel à l'ordre de la version contenant la révision. Si l'arbre de versions s'étend, l'ordre de version croît ainsi que le coût de reconstitution.
Pour diminuer ce coût, nous proposons de calculer le delta de la première révision d'une version branche par rapport à la révision du tronc dont cette version branche est issue. Ceci mène à un nouveau schèma, repris à la figure suivante:

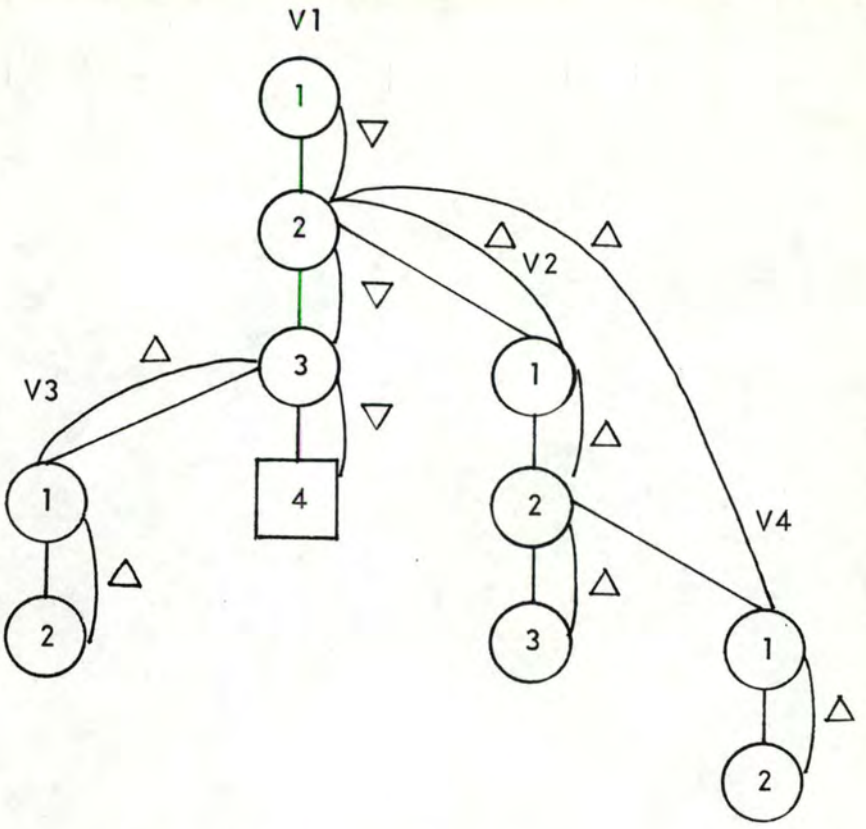


fig.2.11 Solution adoptée.

Les nouveaux deltas calculés sont donc les deltas (V2.1,V1.2) et (V4.1, V1.2).
Ceci présente les avantages suivants:

- à la lecture d'une révision d'une version d'ordre k, nous gagnons k-1 deltas. Ainsi, pour lire V4.2 il ne faut plus que 2 deltas au lieu de 4 ;
- il en est de même au catalogage qui demande la reconstitution d'une révision et le calcul d'un delta dans les deux cas ;
- à la destruction d'une version branche, nous gagnons pour une version d'ordre k, le calcul de x deltas, x étant le nombre de versions d'ordre k+1 créées à partir de cette version. Sur la figure 2.11 , si nous détruisons V2, aucun calcul de delta ne sera nécessaire.

En contrepartie, il y a évidemment une perte de place.
En effet, reprenons uniquement une partie des deltas des figures 2.10 et 2.11 et portons-les sur la figure suivante :

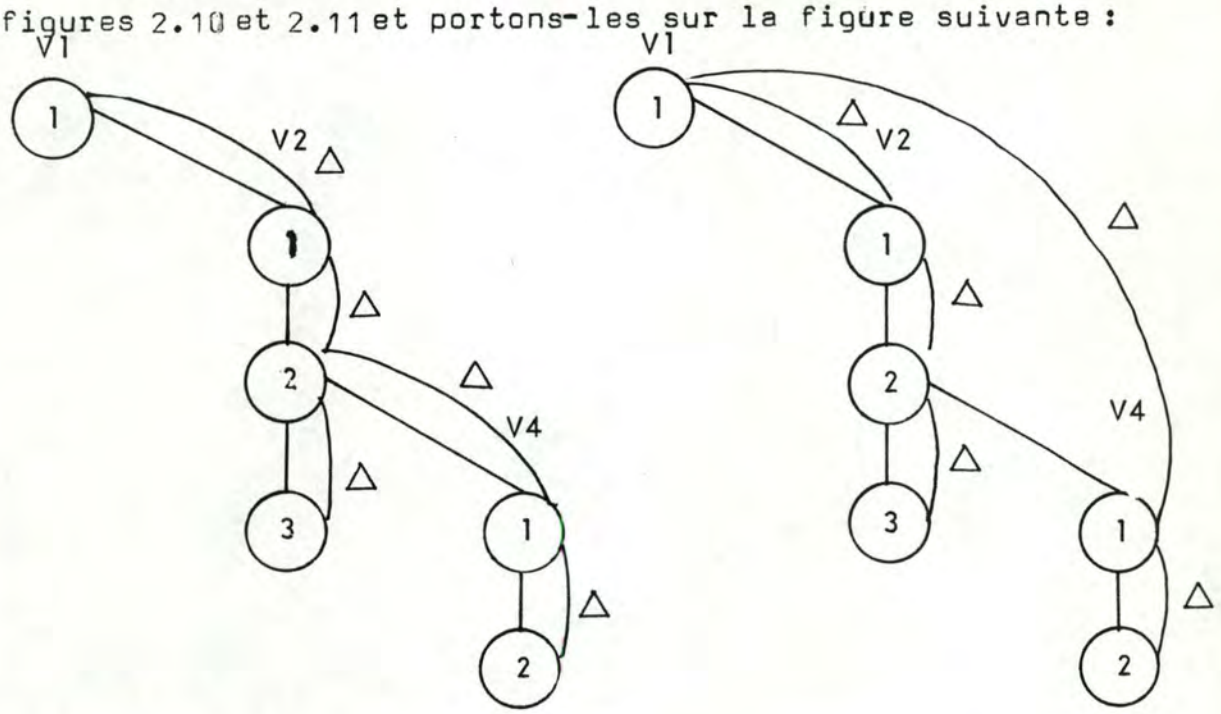


fig.2.12 Différence de reconstitution.

Pour reconstituer V4.1 sur le premier schéma, nous partons du texte de V1.2 auquel nous appliquons successivement les deltas directs (V2.1,V1.2), (V2.2,V2.1), et (V4.1,V2.2).
Sur le second schéma nous appliquons le delta direct (V4.1,V1.2) au texte de V1.2.
La seule différence entre ces deux schémas réside dans le delta(V4.1,V2.2) du premier schéma et le delta(V4.1,V1.2) du second schéma. La perte de place sera donc effective si la place occupée par le delta(V4.1,V1.2) est supérieure à celle occupée par le delta(V4.1,V2.2). Cette inégalité n'étant pas démontrable formellement, elle ne peut être admise qu'intuitivement.

2.1.4. Définition des fonctionnalités du système.

Nous allons présenter dans cette partie les différentes fonctions à remplir par le système, à savoir la mémorisation d'une révision, sa reconstitution et sa destruction.

2.1.4.1. Mémorisation d'une révision.

Cette fonction s'inscrit dans le cadre de la commande de catalogage. L'utilisateur désire en effet envoyer le texte d'une réalisation dans la base et il convient de l'enregistrer sous une forme particulière qui soit en accord avec nos choix de conception.
L'effet de cette mémorisation risque de varier en fonction du type de révision cataloguée. Celle-ci peut en effet appartenir à une arborescence existante de versions, s'adjoindre au tronc ou à une branche, ou encore provoquer la création d'une nouvelle branche de cette arborescence. Elle peut d'autre part provoquer la naissance d'une nouvelle arborescence indépendante. Illustrons ces différentes possibilités sur des exemples en étudiant dans les différents cas le type de mémorisation à produire.

Partons d'une arborescence simple comportant un tronc et une branche.

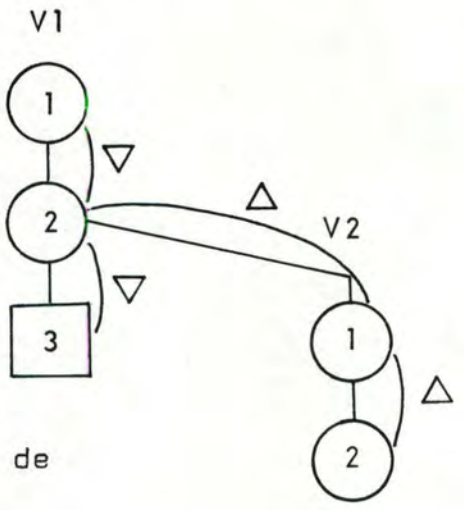


fig.2.13 Arborescence de départ.

1er cas: supposons que l'utilisateur modifie V1.3 et la recatalogue dans V1.4.
Cela conduit à:

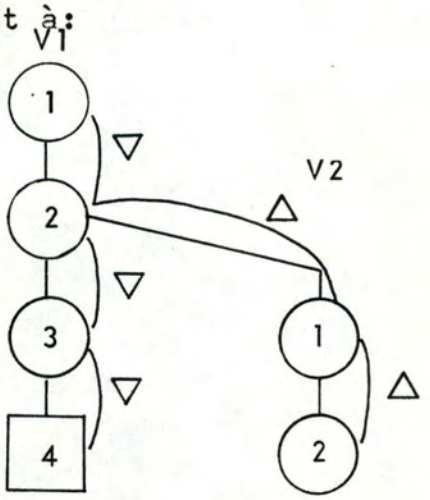


fig.2.14 Premier catalogage.

où la mémorisation de V1.4 s'opère en clair et où V1.3 disparaît au profit d'un delta inverse (V1.3,V1.4).

2e cas: l'utilisateur modifie V2.2 et la recatalogue dans V2.3.
L'arborescence existante devient:

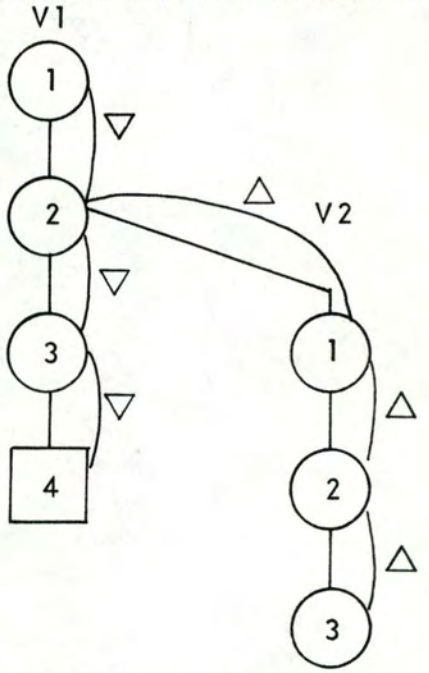


fig.2.15 Second catalogage.

On constate que la mémorisation de V2.3 ne donne pas naissance à une conservation intégrale mais fait apparaître un delta direct (V2.3,V2.2).

3e cas: l'utilisateur modifie V1.3 et la recatalogue dans une nouvelle version qu'il appelle V3. Nous assistons par conséquent à la création d'une branche d'ordre 1.
Dans ce cas, comme lors de la création d'une branche d'ordre k, l'utilisateur doit pouvoir choisir d'intégrer sa révision dans l'arborescence ou de la cataloguer indépendamment de cette dernière, créant ainsi un nouveau tronc.
Cela conduit aux deux schémas suivants:

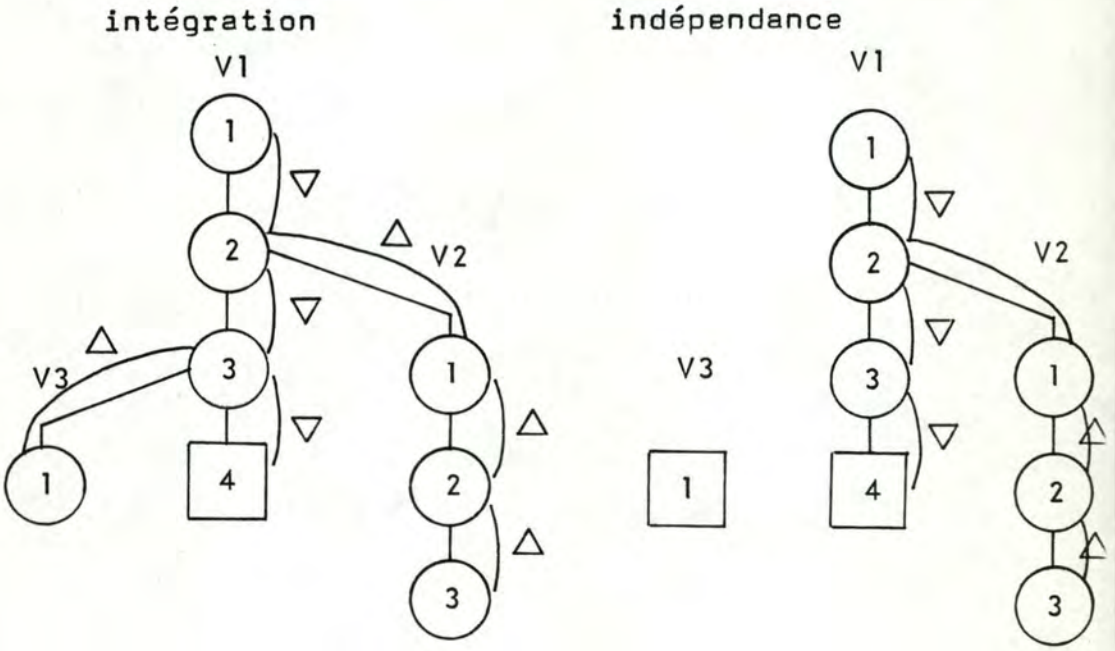


fig.2.16 Intégration ou indépendance.

Dans le cas d'indépendance, la révision V3.1 est conservée intégralement sans aucun lien avec l'arborescence existante, et dans le cas de l'intégration, on constate que la mémorisation de V3.1 se concrétise par le delta direct (V3.1,V1.3).

4e cas: l'utilisateur modifie V2.2 et la recatalogue dans une nouvelle version qu'il appelle V4. Il crée ainsi une branche d'ordre 2 et la remarque du cas précédent reste par conséquent d'application.
Nous n'étudions ici que le cas de l'intégration.

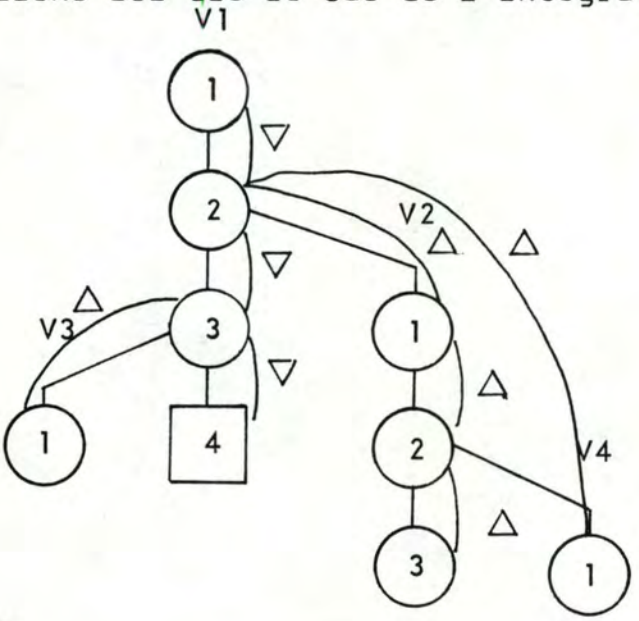
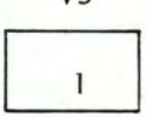


fig.2.17 Création de la version V4.

Nous remarquons que la mémorisation de V4.1 s'effectue par l'intermédiaire d'un delta direct (V4.1,V1.2).

5e cas: L'utilisateur crée une nouvelle version qu'il appelle V5. Elle ne résulte de la modification d'aucune révision existante et doit par conséquent être considérée comme un tronc. Sa mémorisation donne naissance à:



où l'entièreté du texte est conservée.

2.1.4.2. Reconstitution d'une révision.

Le but de cette fonction est de permettre de reconstituer une révision quelconque d'un arbre de versions et ce de manière transparente à l'usager. Celui-ci demande la lecture d'une révision par l'intermédiaire de la commande existante LIRE.
Par exemple, la commande "lire V1.02 fichloc" signifiera "placer la révision 2 de la version V1 dans le fichier local de l'utilisateur nommé fichloc".
Pour réaliser ceci, nous disposons d'une révision gardée en clair et d'un ensemble de deltas comme le montre la figure suivante:

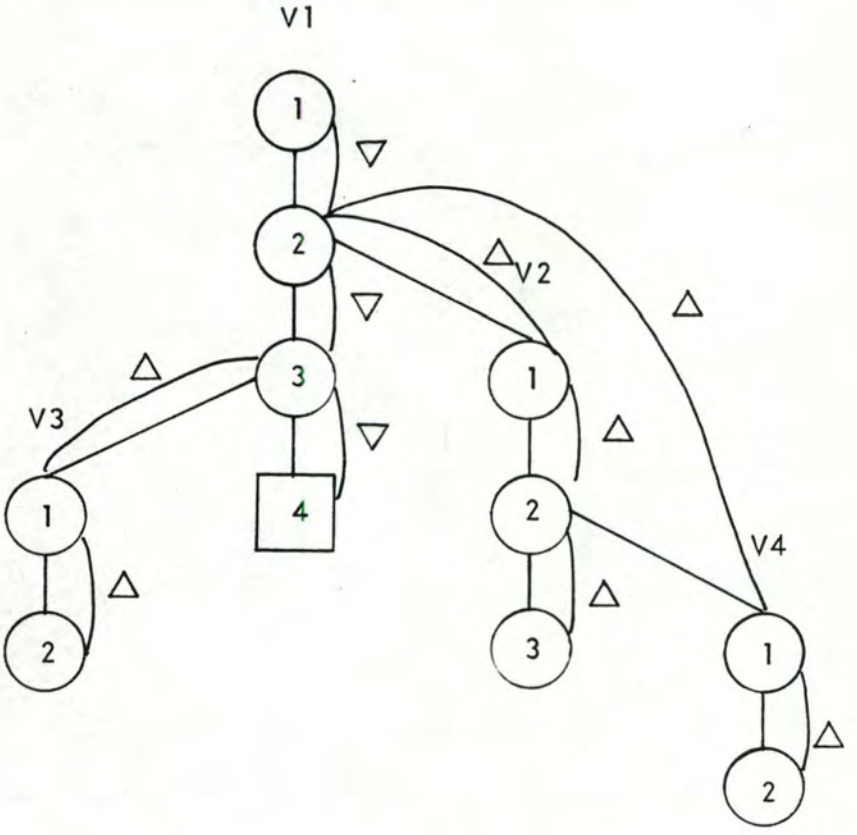


fig.2.18 Rappel de la solution choisie.

Comme nous l'avons vu, la lecture d'une révision d'une version tronc n'est pas identique à celle d'une révision d'une version branche.

(1) Si nous demandons de lire la révision j d'une version tronc sachant qu'il y en a i ($i > j$), nous allons appliquer successivement les deltas (i-1,i), ..., (j,j+1) sur la révision i.
Si i=j, il suffit de lire la dernière révision gardée en clair.

(2) Pour lire la révision j d'une version branche, il faut reconstituer la révision R de la version tronc dont la version branche est issue. Ensuite, il faut appliquer à R les deltas successifs pour atteindre la branche désirée, puis les deltas successifs le long de cette branche pour reconstituer j. Ceci peut se résumer par la suite: delta(1,R), delta(2,1), ..., delta(j,j-1).

2.1.4.3. Destruction de révisions.

Au fil du temps, une arborescence de versions peut devenir volumineuse. Certaines révisions peuvent s'avérer inutiles. Il est donc intéressant de pouvoir détruire des révisions qui ne seront plus jamais utilisées. Il y a deux possibilités:

- soit détruire toutes les révisions antérieures à une révision donnée ;
- soit détruire toutes les révisions de numéro compris entre deux numéros donnés, bornes incluses.

Il est préférable de refuser toute destruction de révisions dans une version tronc car celles-ci peuvent servir à la reconstitution des révisions des versions branches.

Etudions donc les deux types de destruction sur les versions branches:

(1) Destruction des révisions antérieures à une révision:

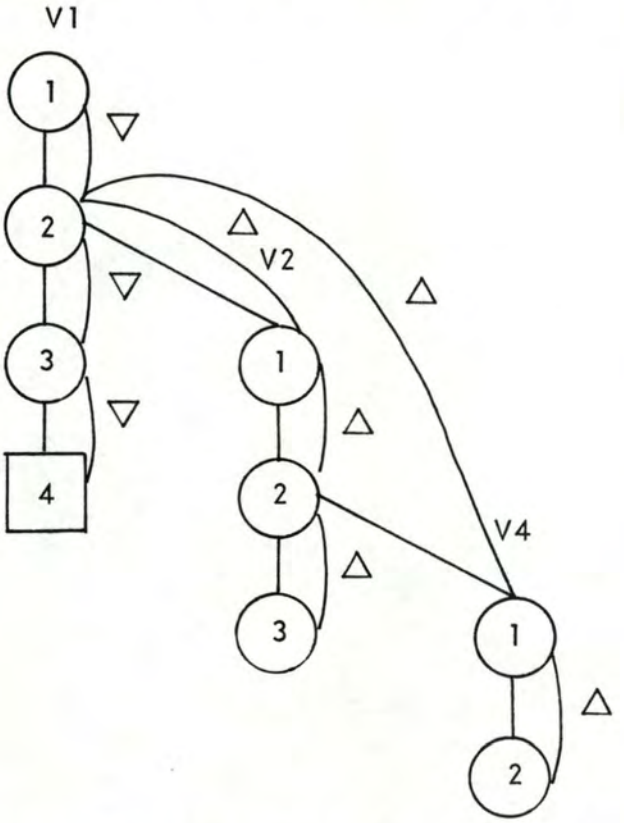


fig.2.19 Destruction antérieure: exemple.

Représentons par R la révision de la version tronc dont la version branche est issue. Pour détruire les révisions antérieures à la révision i, il faut éliminer les deltas suivants: (1,R), (2,1), ..., (i,i-1), et les remplacer par un delta (i,R).

(2) Destruction de révisions dont le numéro est inclus entre deux numéros donnés:
pour détruire les révisions de numéro compris entre i et j inclus ($i < j$), il faut éliminer les deltas suivants: (i,i-1), ..., (j+1,j) et les remplacer par le delta (j+1,i-1).

Rappelons (cfr. §2.1.3.1.) que le delta entre deux révisions non successives, comme le delta (i,R) dans le premier cas ou le delta (j+1,i-1) dans le second cas, est plus volumineux qu'un delta calculé entre deux révisions successives. Ceci signifie que dans certains cas, la destruction de révisions risque d'avoir des répercussions sur le volume disque occupé.

2.1.4.4. Corrections interactives de plusieurs versions.

Dans un arbre de versions, il arrive que certaines erreurs soient communes à plusieurs versions. En cas de détection d'une erreur, la pratique habituelle serait de la corriger via un éditeur et ce pour chaque version. Il nous a dès lors semblé intéressant d'offrir un outil qui permet de répercuter des corrections sur les autres versions concernées et ce de façon interactive c'est-à-dire guidée par l'utilisateur. Une correction automatique (non contrôlée par l'utilisateur) pourrait avoir des effets de bord indésirables à l'exécution et même à la compilation des textes des versions corrigées. Après recherche, nous avons constaté, à l'insu des autres membres de l'équipe, qu'un tel outil existait déjà sur le système Multics, sous la forme d'une commande, le merge-ascii.
Sa fonction de base, entre autres, est de fusionner deux fichiers, le résultat se trouvant sur un troisième fichier. Pour mieux comprendre cette fusion, prenons un exemple: soit deux fichiers X et Y ayant deux blocs de lignes différents A et B que nous schématisons par la figure suivante:

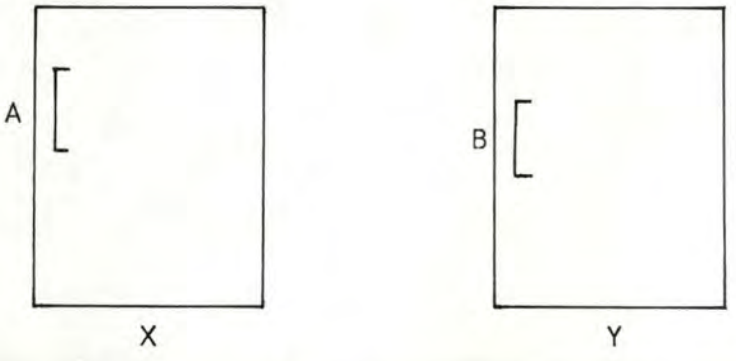


fig.2.20 Fichiers à fusionner.

Les lignes précédant et suivant A dans X sont identiques à celles qui précèdent et suivent B dans Y.
L'exécution du merge-ascii recopie les lignes précédant A dans le fichier des résultats, soit Z. Ensuite, il met dans deux buffers les blocs de lignes A et B. L'utilisateur se trouve alors en mode éditeur qui permet d'exécuter différentes commandes:

- copier un buffer sur le troisième fichier ;
- copier une partie du buffer sur le troisième fichier ;
- imprimer la totalité ou une partie du buffer ;
- introduire de nouvelles lignes dans le troisième fichier ;
- détruire des lignes du troisième fichier ;
- imprimer des lignes du troisième fichier ;
- quitter l'éditeur et continuer la comparaison ;

Une fois les lignes des blocs A et B sélectionnées et recopiées sur le fichier Z par l'utilisateur, celui-ci quitte le mode éditeur. Les lignes communes à X et Y suivant A sont recopiées sur Z.
Ce processus peut s'appliquer plusieurs fois si les fichiers de départ comportent plusieurs blocs de lignes différentes.
Pour plus d'informations sur cette commande, on peut consulter (Mul, 1) et (Mul, 2).

Par conséquent, si nous voulons corriger des erreurs communes à plusieurs versions, il faut d'abord les corriger une première fois avec un éditeur sur le texte d'une de ces versions, et ensuite appliquer le merge-ascii entre cette version corrigée et chaque version à corriger.
Il faut bien préciser que le merge-ascii est un outil disponible sur Multics au même titre que l'éditeur EMACS dont l'utilisateur se sert pour corriger des textes de versions, introduire de la documentation, ...
Cette fonction ne sera donc pas ajoutée à la base Adèle.

2.2. Alternatives de réalisation.

Ce paragraphe est destiné à la présentation de diverses possibilités de mémorisation de deltas. On présente par conséquent des alternatives susceptibles de réaliser le choix de conception présenté précédemment.

Rappelons que le choix de conception relatif à une arborescence de versions peut se schématiser par la figure 2.11 présentée en 2.1.3.2..

La dernière révision de la version tronc est gardée intégralement et les révisions antérieures peuvent être régénérées au moyen de deltas inverses successifs.
En ce qui concerne les branches, la première révision peut être reconstituée grâce au delta direct qui la relie à la révision du tronc dont elle est issue, et les révisions ultérieures sont obtenues au moyen de deltas directs successifs.
La conservation de ces différents deltas nous a conduits directement à effectuer un choix de réalisation.

Trois possibilités nous étaient offertes. La première d'entre elles, où tous les deltas sont implémentés en fichiers indépendants et constituent par conséquent des ensembles de commandes autonomes, peut se symboliser par le schéma suivant:

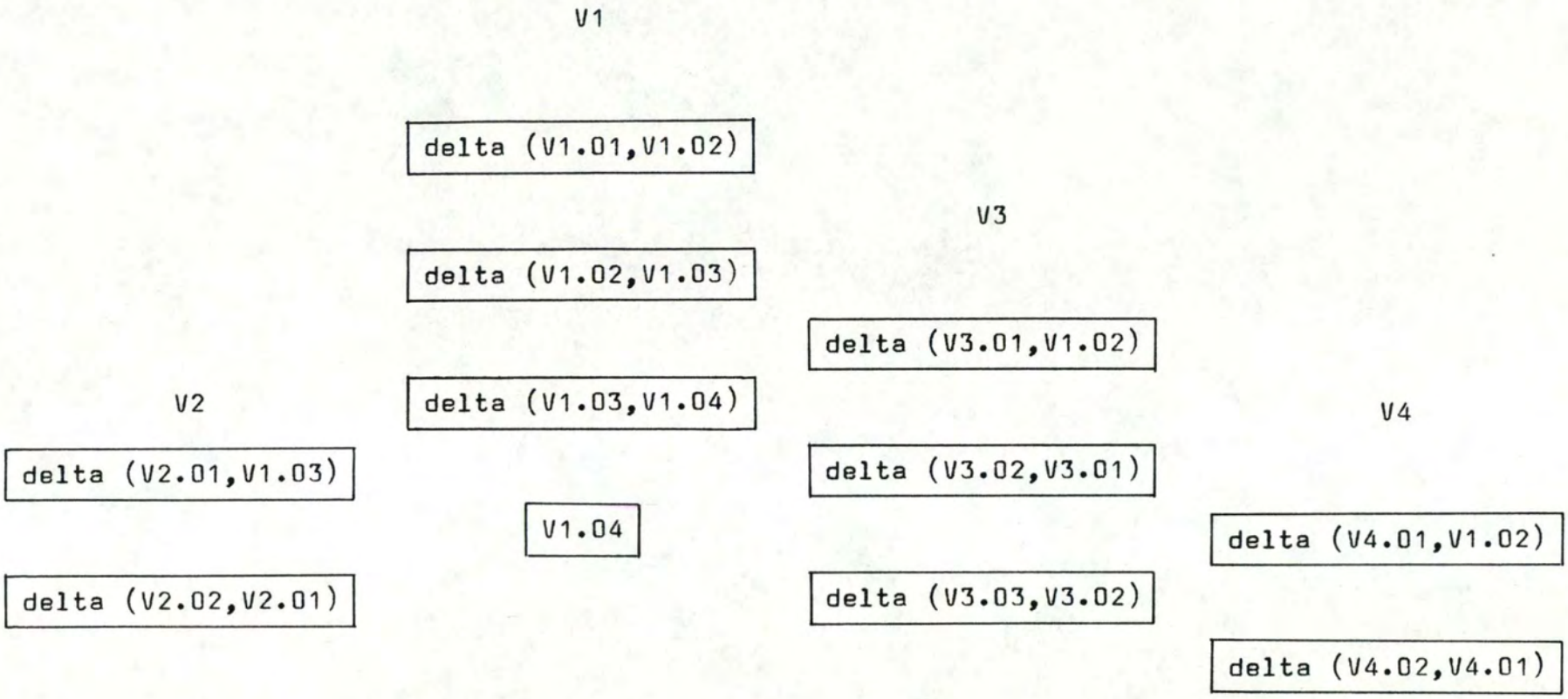


fig.2.21 Deltas séparés.

La seconde possibilité était de concentrer tous les deltas relatifs à une branche particulière à l'intérieur d'un seul et unique fichier. Cette politique pouvait par conséquent se schématiser de la manière suivante:

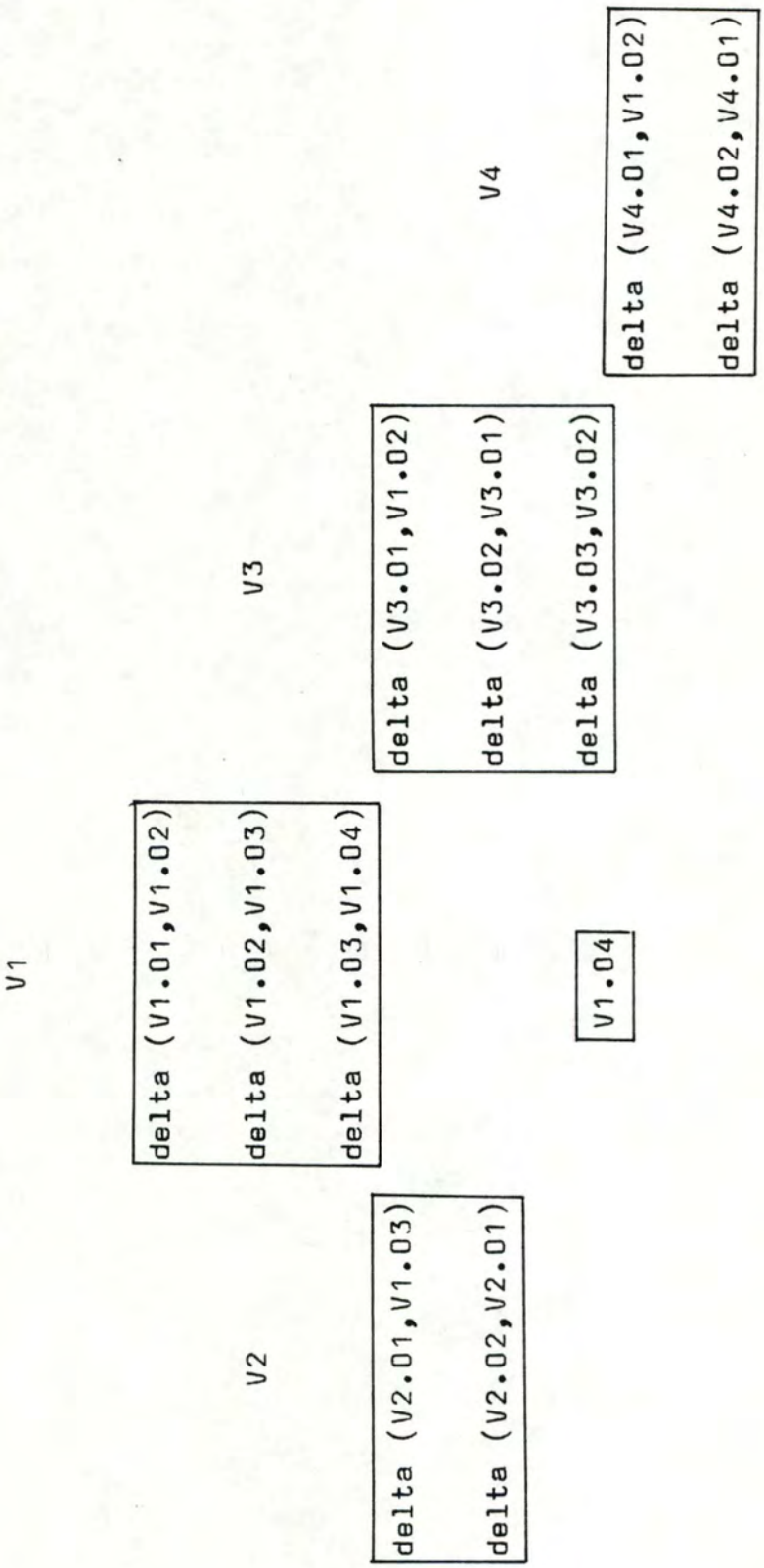


fig.2.22 Deltas groupés.

Une troisième possibilité consiste à reprendre tous les deltas sur un même fichier à la manière de RCS (cfr. I.2).
Nous avons directement écarté cette possibilité du fait de la complexité du fichier qu'elle aurait entraînée.

C'est en fait la seconde possibilité que nous avons adoptée. Notre choix a surtout été guidé par les raisons suivantes:

(1) Nous avons essayé de minimiser autant que possible le temps de reconstitution d'une révision. Par la première méthode de conservation, nous possédons autant de fichiers que de deltas à appliquer pour régénérer une révision. Ceci implique les opérations d'ouverture et de fermeture des différents fichiers, ce qui sous Multics prend un temps non négligeable.

(2) La page (2048 caractères) représente l'allocation minimale d'espace disque sous Multics. Les deltas étant en général petits, la multiplication de petits fichiers que propose la première solution risquait de nous faire perdre une place importante alors que notre but est d'en gagner un maximum.

Il est clair que sous ces deux points de vue, la seconde solution paraît la plus intéressante. Néanmoins, elle présente certains problèmes que les deltas autonomes de la première solution nous auraient permis d'éviter. Ces problèmes résultent du fait que conceptuellement deux deltas différents sont indépendants, alors que dens la deuxième solution, ils apparaissent liés par l'appartenance à une même suite. Il ne faut pas oublier que ces deltas sont constitués de commandes d'édition qui seront exécutées en séquence pour pouvoir régénérer une révision. Pour la plupart des révisions, cette régénération ne nécessitera la mise en oeuvre que d'une partie des deltas appartenant au même ensemble. L'application de tous les deltas d'un ensemble ne sera effectuée que dans le cas d'une reconstitution de la première révision du tronc ou de la dernière

révision d'une branche si celle-ci est issue de la première révision du tronc.
La reconstitution de V1.02 par exemple nécessitera l'application des deltas (V1.03,V1.04) et (V1.03,V1.02) indépendamment du delta (V1.01,V2.01). Il faudrait par conséquent pouvoir extraire un delta de son ensemble mais on se rend rapidement compte qu'une telle extraction nuirait à la rapidité de la reconstitution alors que cette rapidité est justement l'une des raisons qui nous a fait pencher vers la seconde solution.

Nous avons par conséquent cherché une autre façon de contourner le problème. Nous avons décidé de délimiter chaque delta d'une manière telle que l'exclusion de certains d'entre eux soit automatique lors de la reconstitution d'une révision.
Cette délimitation peut s'exprimer de la façon suivante:
si la révision à reconstituer est égale à celle qui vient d'être regénérée par le delta précédent, ne pas appliquer les deltas ultérieurs. Elle consiste donc en quelque sorte en un ordre à l'éditeur, imposant la fin d'un traitement.
Cette délimitation n'est évidemment pas suffisante pour empêcher l'extraction. Telle qu'elle est exprimée, elle induit que les n ($n \geqslant 1$) premiers deltas d'un ensemble sont toujours appliqués. Cette propriété est toujours vraie pour les branches puisqu'une de leurs révisions est toujours obtenue à partir de la révision précédente mais elle s'avère fausse pour le tronc.
La reconstitution de V1.03 ne nécessite l'application que du delta (V1.03,V1.04) et les deux premiers delats n'entrent pas en jeu. Cela nous conduit à enregistrer les différents deltas dans l'ordre inverse pour le tronc de manière à ce que les n premiers deltas de n'importe quel ensemble soient toujours appliqués quelle que soit la révision à regénérer.

Ces différentes considérations nous ont conduits au schéma suivant:

V1

```
delta (V1.03,V1.04)
Si r-à-r = 03, arrêter
delta (V1.02,V1.03)
Si r-à-r = 02, arrêter
delta (V1.01,V1.02)
Si r-à-r = 01, arrêter
```

V2

```
delta (V2.01,V1.03)
Si r-à-r = 01, arrêter
delta (V2.02,V2.01)
Si r-à-r = 02, arrêter
```

V1.04

V3

```
delta (V3.01,V1.02)
Si r-à-r = 01, arrêter
delta (V3.02,V3.01)
Si r-à-r = 02, arrêter
delta (V3.03,V3.02)
Si r-à-r = 03, arrêter
```

V4

```
delta (V4.01,V1.02)
Si r-à-r = 01, arrêter
delta (V4.02,V4.01)
Si r-à-r = 02, arrêter
```

où r-à-r = révision à contrôler

fig. 2.23 Séparateur de deltas.

Les différents fichiers de deltas porteront le nom de la révision 00 de la version à laquelle ils se rapportent; le fichier associé à V1 s'appellera par conséquent

> F> ... >Fn-I-V1.text.00 .

Les liens entre les différentes versions seront conservés grâce aux manuels associés, par l'intermédiaire d'un champ spécial: rev-origine.
Pour l'exemple utilisé, nous aurons l'ensemble des manuels suivants:

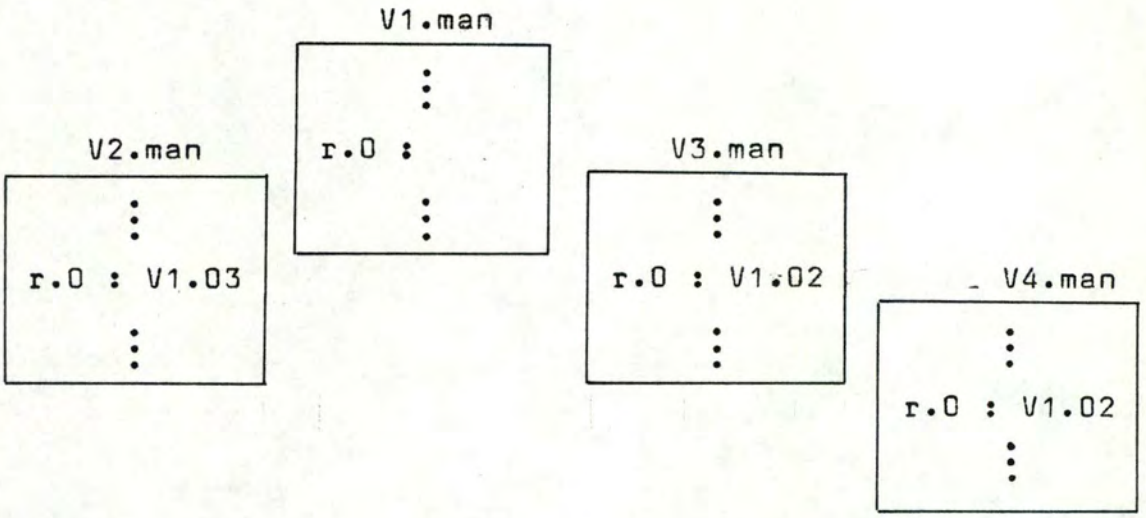


fig.2.24 Manuels associés.

2.3. Conception globale du système et réalisation.

2.3.1. Architecture.

Le schéma présenté ci-après met en évidence les différents composants du système de gestion des deltas. Nous pouvons constater que les trois modules fonctionnels s'intègrent dans la base de la façon suivante:

- le module de destruction de révisions est utilisé par le module général de destruction de la base;

- le module de mémorisation d'une révision est utilisé par celui du catalogage;

- le module de reconstitution d'une révision est utilisé par le module de lecture de la base.

Ces différents modules fonctionnels sont explicités en terme des divers composants qu'ils utilisent. La spécification et la réalisation de chacun d'eux feront l'objet des paragraphes suivants.

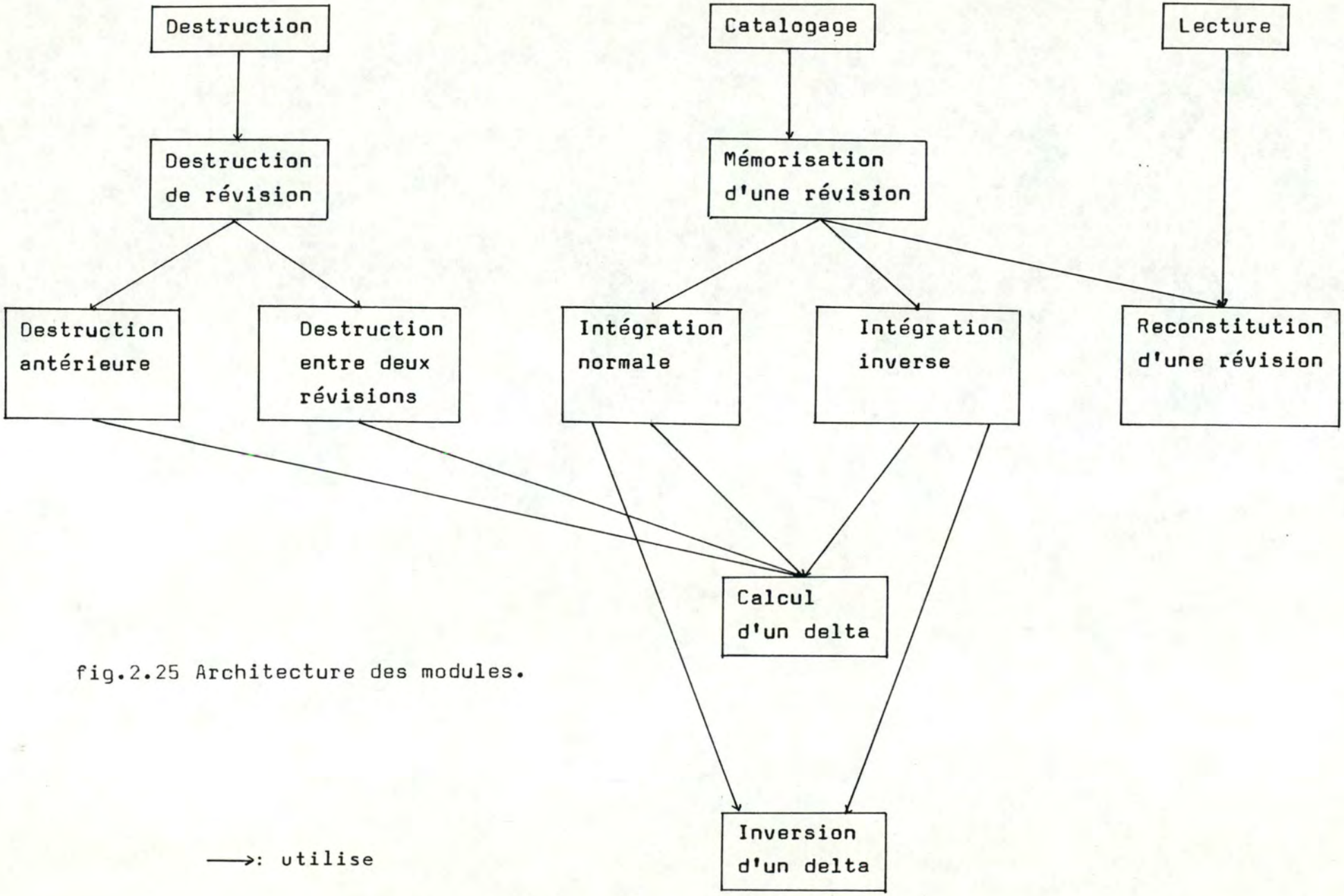


fig.2.25 Architecture des modules.

2.3.2. Calcul d'un delta.

On remarque sur l'architecture que ce module est utilisé par quatre autres composants. Il apparaît donc essentiel et c'est pourquoi nous commençons par sa spécification.

2.3.2.1. Spécification.

Ce module a pour fonction de calculer la différence entre deux textes. La différence calculée, ou delta, est un ensemble de commandes d'édition qui permettra de régénérer l'un des textes à partir du contenu de l'autre. A cette fin, il reçoit en entrée les arguments suivants:

- fich-origine: nom d'un fichier de texte
- fich-destination: nom d'un fichier

précondition:
les deux fichiers doivent exister.

Il produit en sortie un troisième fichier appelé !Bdelta, qui contient le delta généré. Ce fichier répond aux postconditions suivantes:

(i) Suivant la nomenclature adoptée, le delta produit = delta(fich-destination,fich-origine).

(ii) Si les contenus des deux fichiers d'entrée sont identiques, le fichier de sortie contient uniquement les lignes:

"A fich-origine (original)
"B fich-destination (new)

(iii) Si les contenus sont différents, il contient en plus des deux lignes précitées un ensemble de commandes d'édition qui permettront de recréer le contenu de fich-origine à partir du texte de fich-destination.

Il existe quatre commandes différentes:

1) Suppression.

La commande i,jd donne l'ordre de supprimer les lignes dont le numéro d'ordre est compris entre i et j inclus ($i \leqslant j$) dans le fichier sur lequel elle est appliquée.

2) Concaténation.

La commande

```
  !a
ligne 1
  :
  :
ligne n
.
```

permet de concaténer les lignes 1 à n ($n \geqslant 1$) au fichier sur lequel elle est appliquée. Les lignes 1 à n sont concaténées telles qu'elles sont. Cela signifie que si elles contiennent des caractères de contrôle (# pour supprimer le caractère précédent, @ pour supprimer tous les caractères précédents dans la ligne...), ceux-ci n'ont aucun effet et sont insérés comme n'importe quel caractère.

Afin de montrer les problèmes posés par l'utilisation d'une telle commande, étudions le cas suivant:

- Supposons qu'une commande de concaténation d'une ligne quelconque, d'un point puis d'une autre ligne soit nécessaire pour reconstituer un texte.
Une telle commande doit s'écrire:

```
!a
ligne 1
.
ligne 2
.
```

- Le point étant le délimiteur de la commande !a, l'éditeur va considérer le premier . comme délimiteur et aucune concaténation ne se produira.

- Il convient dans ce cas d'utiliser les commandes suivantes:

```
!a          (append 1)
ligne 1     (ligne à insérer)
.           (fin append 1)
a           (append 2)
.           (. à insérer)
\f          (fin append 2)
!a          (append 3)
ligne 2     (ligne à insérer)
.           (fin append 3)
```

La commande "a" est en effet délimitée par \f et le problème exposé est contourné. Si nous n'avons pas utilisé cette commande dans tous les cas, c'est simplement parce qu'elle présente l'inconvénient de laisser tout leur sens aux caractères de contrôle éventuellement contenus dans les lignes à concaténer.

3) Substitution.

La commande

```
i,j!c
ligne 1
  :
ligne n
.
```

permet de changer les ligne i à j du fichier sur lequel elle est appliquée, par les les lignes 1 à n. Elle se délimite également par "." et par conséquent, la remarque de la commande précédente reste d'application.

4) Insertion.

La commande

```
j!i
ligne 1
  :
ligne n
.
```

a pour effet d'insérer les lignes 1 à n avant la ligne j ($j \geqslant 1$) du fichier sur lequel elle est appliquée. La remarque des deux commandes précédentes reste toujours valable.

Dans le cas où les fichiers comparés sont différents, le fichier de sortie possède la structure suivante:

"A fich-origine (original)
"B fich-destination (new)
NL
C1
NL
NL
C2
NL
NL
C3
:
Cn
NL

où les différents Ci ($1 \leqslant i \leqslant n$) représentent chacun un bloc de une ou plusieurs lignes correspondant aux quatre commandes définies. Chaque bloc est séparé du suivant par deux lignes vides contenant chacune l'unique caractère NL (new line). Les différents blocs obéissent à l'observation suivante: les numéros de lignes intervenant dans le bloc Ci sont toujours inférieurs à ceux intervenant dans le bloc Cj si i est inférieur à j. Cette remarque implique que la commande $!a ne peut appartenir qu'au bloc Cn.

2.3.2.2. Exemple.

Nous allons illustrer l'effet du module de calcul d'un delta sur un exemple particulier.

Supposons les deux fichiers suivants:

fichier 1	fichier 2
1 Toto va à l'école	1 Toto va à l'école
2 avec sa mallette.	2 avec son cartable.
3 Il y rencontre Marc	3 Il y rencontre Marc
4 et devient son copain.	4 et devient son copain.
5 Le soir, il rentre	5 Pendant la journée, il travaille.
6 chez lui avec Mireille.	6 Le soir, il retourne
7 Il est tard.	7 chez lui avec Mireille.

où le fichier 1 est le fichier original et le fichier 2 le nouveau fichier.

Il faut donc remplacer les lignes 2 et 6 dans le fichier 2, y supprimer la ligne 5 et y ajouter la ligne "Il est tard" pour retrouver le contenu du fichier 1.

Le calcul du delta entre ces deux fichiers produira ainsi:

```
"A fichier 1 (original)
"B fichier 2 (new)
NL
2,2!C
avec sa mallette.
.
NL
NL
5d
6,6!C
Le soir, il rentre
.
NL
NL
$ !a
Il est tard.
.
NL
```

2.3.2.3. Réalisation.

La réalisation de ce module a été relativement facilitée par la disponibilité d'un outil existant sous Multics: il s'agit d'un module complexe (CPA) qui permet de comparer jusqu'à cinq fichiers par rapport à un fichier original. Il fournit dans un fichier de sortie les différences entre chacun des fichiers d'entrée et le fichier original. Si un caractère d'une ligne diffère, il considère toute la ligne comme une différence.
La performance de ce logiciel nous a incités à le récupérer et à l'adapter à nos propres besoins. Bien qu'il fournisse de nombreuses options, nous l'utilisons toujours sous forme particulière suivante:

```
CPA  fichA  fichB  -minlines 1  -ntt  -of fichC
```

où fichA représente le fichier original et fichB le nouveau fichier. Les autres paramètres représentent des options particulières:

- minlines 1: signifie qu'il suffit de trouver une seule ligne équivalente dans les deux fichiers pour délimiter une différence.

- ntt: signifie qu'on ne désire pas un titre de totalisation de différences en sortie.

- of fichC: signifie que les différences seront consignées sur le fichier fichC. Si cette option n'est pas utilisée, les différences sont visualisées directement à l'écran.

A l'issue de l'exécution de ce type de commande, fichC contient les renseignements suivants:

"A fichA (original)
"B fichB (new)
NL
bloc 1
NL
NL
bloc 2
NL
NL
:
bloc n
NL

avec $n \geqslant 0$ et où les différents blocs i correspondent chacun à l'un des 4 renseignements suivants:

1) Ai
:
Aj
Changed by B to:
B1
:
Bk

2) Ai
:
Aj
Deleted at end.

3) B1
:
Bk
Inserted in B, preceding:
Ai

4) Ai
:
Aj
Deleted by B, preceding:
B1

Etant donné notre objectif de produire une macro permettant de reconstituer fichA à partir de fichB, ces quatre blocs ont été modifiés de la façon suivante:

1) Ce bloc signifie que les lignes i à j de fichA ont été substituées par les lignes 1 à k dans fichB. Il convient par conséquent de remplacer ces dernières par les lignes i à j de fichA pour retrouver le fichier original.
Ceci s'opère par la commande:

```
1,k!C
Ai
:
Aj
.
```

2) Cette information signifie que les lignes i à n (n étant la dernière ligne de fichA) ont été supprimées du fichier original. Il convient par conséquent de les réinsérer par la commande:

```
$ !a
Ai
:
Aj
.
```

3) Ce bloc exprime que les lignes 1 à k de fichB ont été insérées avant la ligne i de fichA. Il convient donc de les supprimer par la commande:

```
1,kd
```

4) Ce dernier bloc renseigne que les lignes i à j de fichA, qui précédaient la ligne 1 de fichB, ont été suppri-

mées. Il suffit donc de les réinsérer par la commande:

```
l!i
Ai
:
:
Aj
.
```

Rappelons que, pour les blocs 1), 2) et 4), si une des lignes i à j contient le seul caractère ".", il faut la remplacer par:

```
.
a
.
\ f
!a
```

2.3.3. Inversion d'un delta.

En se référant à l'architecture, on peut remarquer que ce module n'utilise rien et qu'il est utilisé par les deux modules essentiels du système, à savoir l'intégration normale pour les deltas directs, et l'intégration inverse pour les deltas inverses. C'est précisément le fait qu'il est utilisé par des modules différents qui a motivé celui d'en faire un module indépendant.

2.3.3.1. Spécification.

Les deltas tels que construits à la section 2.3.2. ne permettent pas toujours de reconstituer un texte sans erreur. L'exécution de certaines commandes risque en effet de modifier les numéros de lignes intervenant dans les commandes ultérieures. Cela provient du fait que les différentes commandes sont enregistrées dans un ordre strictement croissant des numéros de lignes et qu'elles sont exécutées en séquence. Afin d'illustrer cet inconvénient, considérons l'exemple suivant:

fichier 1	fichier 2
1 Pierre est un garçon	1 Pierre est un garçon
2 belge.	2 français.
3 Jean est français et	3 Il joue avec Lise.
4 joue avec Pierre.	4 Jean est français et
	5 joue avec Pierre.
	6 Marcel est malade.

où le fichier 1 est le fichier original et le fichier 2 le nouveau fichier.

Le delta calculé entre ces deux textes équivaut à:

```
"A fichier 1  (original)
"B fichier 2  (new)
NL
2,2!c
belge.
.
NL
NL
3d
NL
NL
6d
NL
```

L'application des deux premières commandes sur le fichier 2 conduit à:

```
1 Pierre est un garçon
2 belge.
3 Jean est français et
4 joue avec Pierre.
5 Marcel est malade.
```

On remarque par conséquent que la troisième commande (6d) ne peut s'appliquer, la ligne 6 ayant disparu. Le but du présent module est de pallier à ce genre d'inconvénient. On comprend aisément qu'il va s'agir d'inverser l'ordre des commandes de manière à ce que le delta soit trié dans un ordre strictement décroissant des numéros de lignes.
Le module peut se définir comme suit:

- entrée: fich-delta;

- précondition: le fichier fich-delta doit être structu-

ré de la façon suivante:

"A fichier-original (original)
"B fichier-destination (new)
NL
C1
NL
NL
C2
.
.
Cn
NL

avec $n \geqslant 0$.

- sortie: fich-delta;

- postcondition: le fichier fich-delta est réorganisé de la façon suivante:

"A fichier-original (original)
"B fichier-destination (new)
NL
Cn
NL
NL
Cn-1
.
.
C1
NL

avec $n \geqslant 0$.

Pour l'exemple considéré au début de la spécification, nous

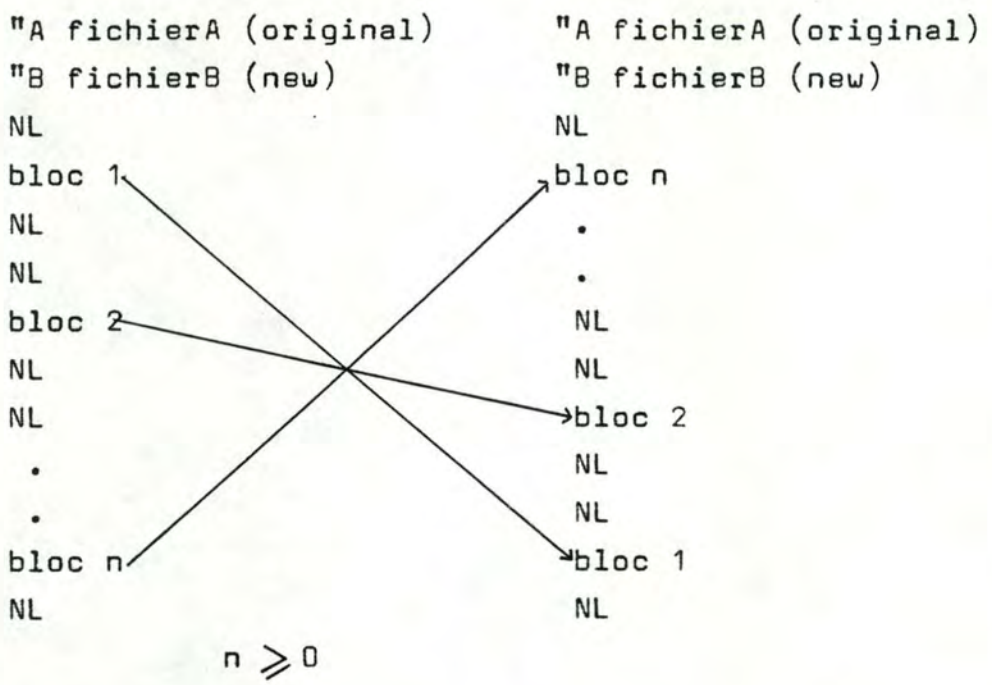
obtenons:

```
"A fichier 1 (original)
"B fichier 2 (new)
NL
6d
NL
NL
3d
NL
NL
2,2!c
belge.
.
NL
```

On peut facilement vérifier que l'application de ces différentes commandes sur le contenu du fichier 2 produit le texte du fichier 1. Ceci peut être démontré en toute généralité par le fait que l'exécution d'une commande sur des lignes de numéro inférieur ou égal à j ne peut modifier que les numéros de lignes supérieurs à j. Et vu que les différentes commandes sont réarrangées dans un ordre strictement décroissant des numéros de lignes, l'exécution d'une commande sur la ligne j exclut l'exécution ultérieure d'une commande sur une ligne de numéro supérieur à j.

2.3.3.2. Réalisation.

Nous avons vu au paragraphe précédent que le fichier des deltas possédait une structure composée d'une entête suivie de 0 à n blocs de taille indéfinie séparés par deux lignes vides. Le but de l'algorithme est d'inverser les blocs de la manière présentée ci-après:



Il est évident qu'une telle opération peut s'avérer complexe et coûteuse en Pascal où les techniques de gestion de fichier ne sont pas toujours des plus agréables. C'est pourquoi cette inversion est implémentée par une macro Ted. Cet éditeur, permettant de travailler simultanément sur plusieurs buffers, facilite considérablement la transformation dont le principe général est le suivant:

- lire le delta dans le buffer 1;
- copier les deux premières lignes (l'entête) dans le buffer 2;
- tant qu'il existe des blocs
  - isoler le premier bloc;
  - copier le premier bloc après la ligne 2 du buffer 2;
  - détruire le bloc copié dans le buffer 1;
- recopier le delta contenu dans le buffer 2;

L'unique problème est de pouvoir isoler un bloc. Ce problème est résolu par le fait que tout bloc possède un délimiteur bien spécifique.

Les blocs !a, !c et !i possèdent le délimiteur ".NLNL". Du point de vue de l'éditeur, ce délimiteur ne correspond pas à la ligne "." suivie de deux lignes vides mais bien à la ligne "." composée des caractères ".NL" suivie de la première ligne vide.
Le bloc d quant à lui est délimité par "NLNL" où le premier NL représente le dernier caractère de la ligne d et le second la première ligne vide.

L'algorithme peut par conséquent se compléter de la façon suivante:

{ !Bdelta contient uniquement les deux lignes d'entête
ou il contient ces deux lignes et une série de bloc i
($1 \leqslant i \leqslant n$) où: - le bloc n est suivi d'une ligne vide
- le bloc 1 est séparé de l'entête par une ligne vide
- le bloc i ($i \neq 1$ et $i \neq n$) est séparé du bloc i+1 par deux lignes vides }.

- LIRE !Bdelta dans le buffer 1;
- SE POSITIONNER en ligne 1;
- COPIER les deux premières lignes dans le buffer 2;
- DETRUIRE les deux premières lignes dans le buffer 1;

(la ligne courante = la première ligne si elle existe)
- TANT QU'il existe des lignes dans le buffer 1
    (ligne courante = ligne vide suivant l'entête ou la seconde ligne vide suivant un bloc = la première ligne du buffer 1)
    (précondition $\Rightarrow$ ligne 2 est une ligne de commande)
    SI (ligne 2 du buffer 1) $\ni$ a ou i ou c ALORS
        - RECHERCHER la chaîne ".NLNL";
        (ligne courante = première ligne vide suivant le bloc)
    SINON
        - RECHERCHER la chaîne "NLNL";

```
                ( ligne courante = première ligne vide sui-
                  vant le bloc d )
                COPIER les lignes 1 à ligne courante APRES la
                       ligne 2 du buffer 2;
                DETRUIRE les lignes copiées, dans le buffer 1;
                ( ligne courante = première ligne du buffer 1
                  si elle existe )
- ECRIRE le buffer 2 sur !Bdelta;
( postcondition )
```

2.3.4. Intégration inverse.

Ce module s'insère dans l'architecture de la façon suivante: il est utilisé par le module fonctionnel de mémorisation d'une révision, et il utilise les deux modules définis dans les deux sections précédentes, à savoir celui de calcul d'un delta et celui de son inversion. Il correspond au concept de delta inverse.

2.3.4.1. Spécification.

Ce module est utilisé lorsqu'il s'agit de mémoriser une révision appartenant au tronc de l'arborescence (cfr. 2.1.4.1).
Il s'occupe de la mise à jour du fichier des deltas associé.
Il se définit de la manière suivante:

- entrées: fich-original: nom du fichier original;
  fich-destination: nom du fichier destination;
  fich-delta: nom du fichier des deltas à mettre à jour;
  no-révision;

- préconditions:
  - le fichier des deltas possède la structure suivante:

```
delta (n-1,n)
si r-à-r = n-1   stop
delta (n-2,n-1)
si r-à-r = n-2   stop
      .
      .
delta (k,k+1)
si r-à-r = k     stop
```

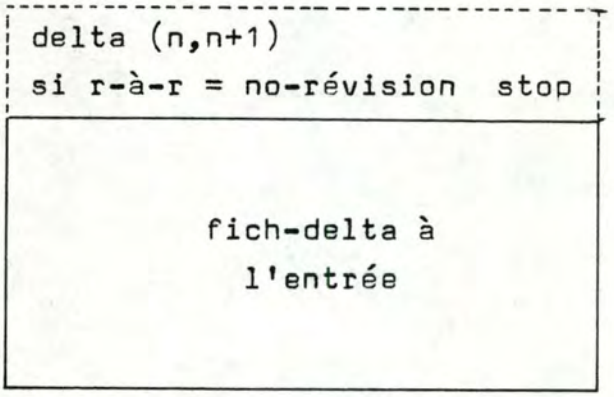
avec $n > 1$ et $1 \leqslant k < n-1$

$n = 1$ ssi le fichier des deltas est vide.

- le fichier original contient le texte de la révision n du tronc;
- le fichier destination contient le texte de la révision n+1;
- no-révision = n;
- le nom du fichier des deltas équivaut au nom de la révision 00 du tronc:

- sortie: fich-delta modifié;

- postcondition:
  - la modification se résume par la figure suivante:



avec $n \geqslant 1$

2.3.4.2. Réalisation.

L'algorithme à mettre en oeuvre doit pouvoir concaténer un delta en tête d'un fichier. Il doit d'autre part utiliser une commande système (CPA) pour effectuer le calcul du delta et une macro Ted pour l'inverser. Une macro ne pouvant être appelée que par une exec-com, cela nous a incités à implémenter ce module sous la forme d'une commande.
Cette exec-com doit:

- calculer le delta entre fich-original et fich-destination;

- inverser le delta calculé;
- intégrer le delta au fichier fich-delta;

Elle doit d'autre part veiller à éviter tout message d'erreur si le fichier des deltas est vide.
Ceci implique la structure générale suivante:

```
CPA fich-original  fich-destination  -minlines 1  -ntt
                                     -of !Bdelta
{ !Bdelta contient le delta qui permettra de reconstituer fich-original (révision n) à partir de fich-destination (révision n+1). Il contient donc bien le delta (n,n+1) }
SI fich-delta est vide ALORS
   traitement-vide
SINON
   traitement-non-vide;
```

où traitement-vide et traitement-non-vide diffèrent très peu.
Traitement-non-vide va devoir:

- inverser le delta contenu dans !Bdelta;
- concaténer le séparateur de delta;
- concaténer le fichier fich-delta;
- recopier l'ensemble sur fich-delta;
- détruire !Bdelta;

Cela se fait facilement dans une exec-com:

```
- INVOQUER Ted;
- LIRE !Bdelta dans le buffer 1;
- APPELER la macro inverseur;
( delta (n,n+1) dans le buffer 2 )
- CONCATENER  "zif P = no-révision" au buffer 2;
( delta (n,n+1) et séparateur dans le buffer 2 )
(1) - LIRE fich-delta dans le buffer 2;
- RECOPIER le buffer 2 dans fich-delta;
- QUITTER Ted;
- DETRUIRE !Bdelta;
( postcondition )
```

Traitement-vide est identique si ce n'est que la ligne (1) n'existe pas. Une telle redondance est nécessaire afin d'éviter que cette ligne ne provoque une erreur. Il est d'autre part impossible de mettre les commandes précédentes en initialisations communes. En effet, ceci nécessiterait:

- d'invoquer Ted;
- d'effectuer les commandes précédant (1);
- de quitter Ted;
- de tester si le fichier des deltas est vide;
- de réinvoquer Ted;

.
.
.

L'unique problème provient du fait qu'après avoir quitté Ted, le contenu des buffers est perdu et que toute invocation ultérieure utilisera des buffers réinitialisés.

2.3.5. Intégration directe.

Tout comme le module précédent, ce module est utilisé par le module fonctionnel de mémorisation d'une révision et il utilise ceux du calcul et de l'inversion d'un delta. Il correspond au concept de delta direct.

2.3.5.1. Spécification.

Ce module est utilisé lorsque l'usager désire cataloguer une révision appartenant à une branche d'une arborescence. Il s'occupe de la mémorisation sous forme condensée de la révision en mettant à jour le fichier des deltas associé à la branche concernée, et peut se définir de la façon suivante:

- entrées: fich-original: nom du fichier original;
  fich-destination: nom du fichier destination:
  fich-delta: nom du fichier des deltas à mettre à jour:
  no-révision:

- préconditions:
  - le fichier des deltas possède la structure suivante:

```
delta (k,révision-tronc d'origine)
si r-à-r = k  stop
delta (k+1,k)
si r-à-r = k+1  stop
      .
      .
delta (n,n-1)
si r-à-r = n  stop
```

avec $1 \leqslant k < n$

si $\begin{cases} k = 0, \text{ le fichier des deltas est vide} \\ n = 1 \end{cases}$

- si n = 1:
  - le fichier destination contient le texte de la révision n (=1) de la branche concernée.
  - le fichier original contient le texte de la révision du tronc d'où provient la branche.
  - no-révision = 1

- si n > 1:
  - le fichier destination contient le texte de la révision n+1 de la branche concernée.
  - le fichier original contient le texte de la révision n de la branche concernée.
  - no-révision = n+1

- fich-delta = nom-version-branche.text.00

- sortie: fich-delta modifié;

- postcondition:
  le fichier des deltas résultant possède l'une des structures suivantes:

n = 1

delta (1, révision-tronc d'origine) si r-à-r = 1 stop

n > 1

fich-delta à l'entrée
delta (n+1,n) si r-à-r = n+1 stop

2.3.5.2. Réalisation.

Tout comme l'intégration inverse et pour les mêmes raisons, ce module est implémenté par une exec-com. Il convient de réaliser un delta (n+1,n), c'est-à-dire un delta qui permettra de reconstituer le fichier destination à partir du fichier original contrairement à l'intégration inverse qui calcul un delta permettant de régénérer le fichier original à partir du fichier destination.
Il convient par conséquent d'invoquer la commande

```
CPA  fich-destination  fich-original  -ntt  -minlines 1
                                      -of !Bdelta
```

pour que !Bdelta contienne le delta (n+1,n).
Le traitement est identique à celui de l'intégration inverse si le fichier des deltas est vide. Dans le cas contraire, il convient d'effectuer les opérations suivantes:

- INVOQUER Ted;
- LIRE !Bdelta dans le buffer 1;
- APPELER la macro inverseur;
(delta (n+1,n) dans le buffer 2)
- CONCATENER "zif P = no-révision" au buffer 2;
- LIRE fich-delta dans le buffer 3;
- COPIER le buffer 2 en fin du buffer 3;
- RECOPIER le buffer 3 dans fich-delta:
- QUITTER Ted;
- DETRUIRE !Bdelta;

2.3.6. Mémorisation d'une révision.

Ce module fonctionnel est invoqué par la procédure de catalogage. Il utilise les modules d'intégration inverse et directe, et celui de reconstitution d'une révision. Les premiers s'occupent de constituer le delta et le second de régénérer une révision si cela s'avère nécessaire.

2.3.6.1. Spécification.

Ce module est utilisé lors du catalogage d'une révision dans la base. L'utilisateur a frappé la commande CATAL nombase fichloc (options)
où nombase représente le nom d'une version et fichloc celui d'un fichier local contenant la révision à cataloguer.
La procédure de catalogage a déjà invoqué la mise à jour globale des historiques et se termine en invoquant le module de mémorisation pour intégrer la révision concernée dans la base.

Rappelons qu'à l'issue des commandes,

(t1) LIRE v1.01 fichloc
(t2) (modifications de fichloc)
(t3) CATAL v1.01 fichloc

fichloc contient la révision v1.01 modifiée et le nom complet de cette révision est connu grâce à la gestion des historiques. (t1),(t2) et (t3) représentent des périodes successives à intervalle quelconque (jours,semaines ...).
Les actions préalables à la mémorisation d'une révision peuvent se schématiser de la manière suivante:

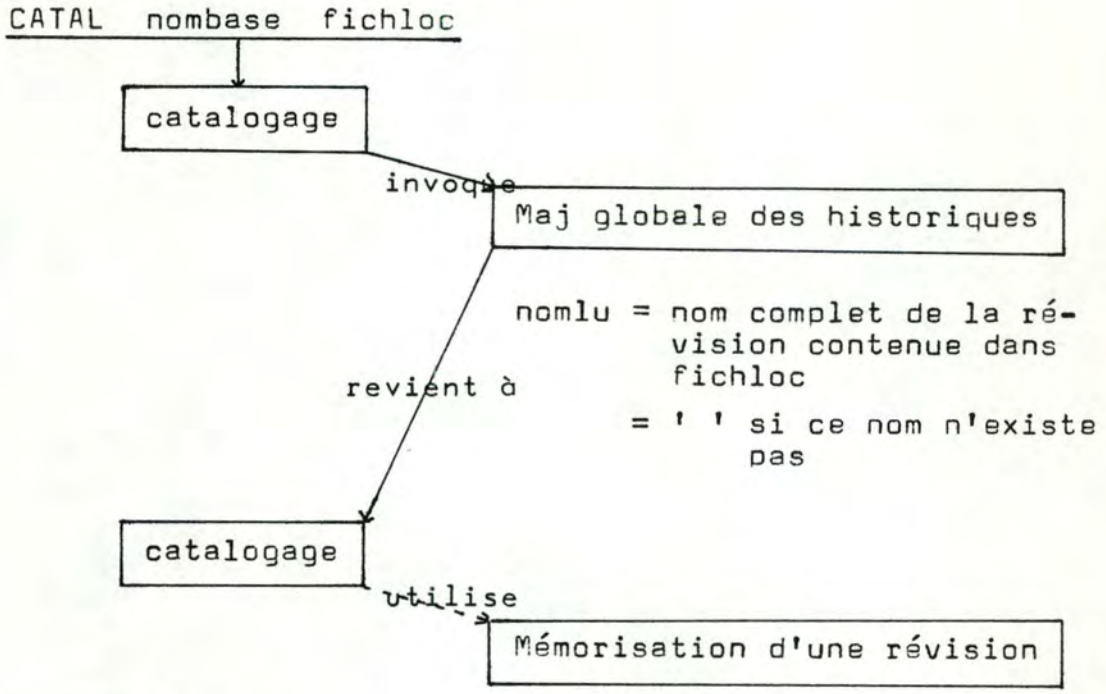


fig.2.26 Enchaînement.

D'autre part, le manuel d'une version permet de différencier les versions branches des versions troncs. Le champs rev-origine égale ' ' si la version est un tronc et il est égal au nom complet de la révision du tronc dont est issue la version si celle-ci est une branche d'ordre quelconque.
Avant de définir le module, rappelons quelques conventions de désignation:

si un nom d'objet de la forme
>F1>F2> ... > Fn-I-V.text.i
est égal à X
alors nomlocal(:X) = >F1>F2> ... >Fn-I-V
no-révision(:X) = i

Le module de mémorisation se définit dès lors de la façon suivante:

- entrées:
  - nombase: nom complet de la révision dans laquelle il faut cataloguer;

- fichloc: nom du fichier local contenant la révision à cataloguer;
- origine: nom complet de la révision contenue dans fichloc;
- ptman : référence de manuel;

- préconditions:
  - ptman doit référencer le manuel associé à l'objet nomlocal(:nombase);
  - origine = ' ' si le nom de la révision contenue dans fichloc est inconnu;
  - si nomlocal(:origine)≠nomlocal(:nombase) et origine≠' '
    alors no-révision(:nombase)='01' ;
  - rev-origine(:nombase)=' ' ssi nomlocal(:nombase) est un tronc:
  - rev-origine(:nombase)≠' ' ssi nomlocal(:nombase) est une branche:

- sorties:
  - si nomlocal(:nombase) ≠ nomlocal(:origine) et origine ≠ ' '
    alors le manuel de nom nomlocal(:nombase).man est modifié;
  - copie intégrale de fichloc dans nombase ou mise à jour du fichier des deltas associé à la version nomlocal(:nombase) et portant le nom nomlocal(:nombase).text.00 ou l'un et l'autre (voir ci-après pour l'explicitation de ces trois possibilités):
  - si no-révision(:nombase) ≠ '01' et nomlocal(:nombase) est un tronc
    alors la révision précédemment gardée en clair est détruite:

- postconditions:
  (1) le manuel nomlocal(:nombase).man est modifié de la façon suivante:

(1.0) le champs rev-origine est égal au nom complet de la révision du tronc d'où provient nombase. Nombase représente la première révision d'une branche d'ordre k;

(1.1) si k>1
alors rev-origine = rev-origine du manuel associé à la version nomlocal(:origine) qui représente une branche d'ordre k-1;

(1.2) si k=1
alors rev-origine = origine;

(2) fichloc est copié physiquement dans nombase ssi
origine = ' ' et no-révision(:nombase) = '01'
ou origine ≠ ' ' et no-révision(:nombase) = '01'
et l'utilisateur désire enregistrer sa révision indépendamment de l'arborescence contenant origine
ou la version nomlocal(:nombase) est un tronc;

(3) le fichier des deltas associé à nomlocal(:nombase) est modifié ssi
no-révision(:nombase) ≠ '01'
ou no-révision(:nombase) = '01' et origine ≠ ' '
et l'utilisateur désire intégrer sa révision dans l'arborescence contenant la révision origine;

(3.1) il est modifié d'une manière "inverse" ssi
no-révision ≠ '01' et la version nomlocal(:nombase) est un tronc.
La modification s'exprime par la postcondition de l'intégration inverse (cfr. 2.3.4).
Le delta à calculer et intégrer est, suivant la nomenclature employée, le delta(révision précédente de nombase, nombase).

(3.2) il est modifié d'une façon "directe" ssi
no-révision ≠ '01' et la version

nomlocal(:nombase) est une branche
ou no-révision = '01' et origine ≠ ' '
et l'utilisateur désire intégrer sa
révision dans l'arborescence contenant
origine.
La modification s'exprime par la postcondition de l'intégration directe (cfr. 2.3.5.).
Le delta à calculer et intégrer varie suivant le no de révision de nombase et l'ordre de la branche contenant nombase:

$o_{r_{d_{r_e}}}$ révis.	=1	1
='01'	delta(nombase,origine)	delta(nombase,X)
'01'	delta(nombase,Y)	delta(nombase,Y)

où X représente la révision du tronc d'où provient la branche,
Y représente la révision précédente de la branche.

2.3.6.2. Construction de l'algorithme.

Remarquons d'abord que plusieurs postconditions seraient simplifiées si nous avions la propriété P:

si origine ≠ ' ' et
no-révision(:nombase) = '01' et
l'utilisateur désire enregistrer sa révision indépendamment de l'arborescence contenant origine
(p) alors
no-révision(:nombase) = '01' et
origine = ' '

ainsi que la propriété Q:

si origine = ' ' et
no-révision(:nombase) = '01' et
l'utilisateur désire intégrer sa révision dans l'arborescence contenant origine;
(Q) alors
no-révision(:nombase) = '01' et
origine = ' ';

La postcondition (2) deviendrait alors:

- fichloc est copié physiquement dans nombase ssi
origine = ' ' et no-révision = '01'
ou la version nomlocal(:nombase) est un tronc;

et la postcondition (3) serait:

- le fichier des deltas associé à nomlocal(:nombase) est modifié ssi
no-révision(:nombase) ≠ '01'
ou no-révision(:nombase) = '01' et origine ≠ ' ';

- il est modifié d'une façon "directe" ssi
no-révision ≠ '01' et la version nomlocal(:nombase) est une branche
ou no-révision = '01' et origine ≠ ' ';

La première étape de l'algorithme consiste dès lors à valider les deux propriétés, de la manière suivante:

```
SUFFIXE (nombase, préfixe, suffixe);
SUFFIXE (suffixe, préfixe, no-révision);
SI origine ≠ ' ' ET no-révision = '01'  ALORS
    IMPRIMER 'Désirez-vous créer une version indépendante (o-n)?';
    REPETER
        DEMANDE réponse;
    JUSQUE réponse = 'o' ou 'y' ou 'n' ;
    SI réponse = 'o' OU 'y' ALORS
        origine = ' ';
( P et Q )
( no-révision = no-révision(:nombase) )
```

La suite de l'algorithme consiste à analyser la situation dans laquelle on se trouve et à mener les actions nécessaires pour atteindre les postconditions. Notons que cinq situations différentes s'avèrent possibles. Elles sont résumées à la figure de la page suivante.

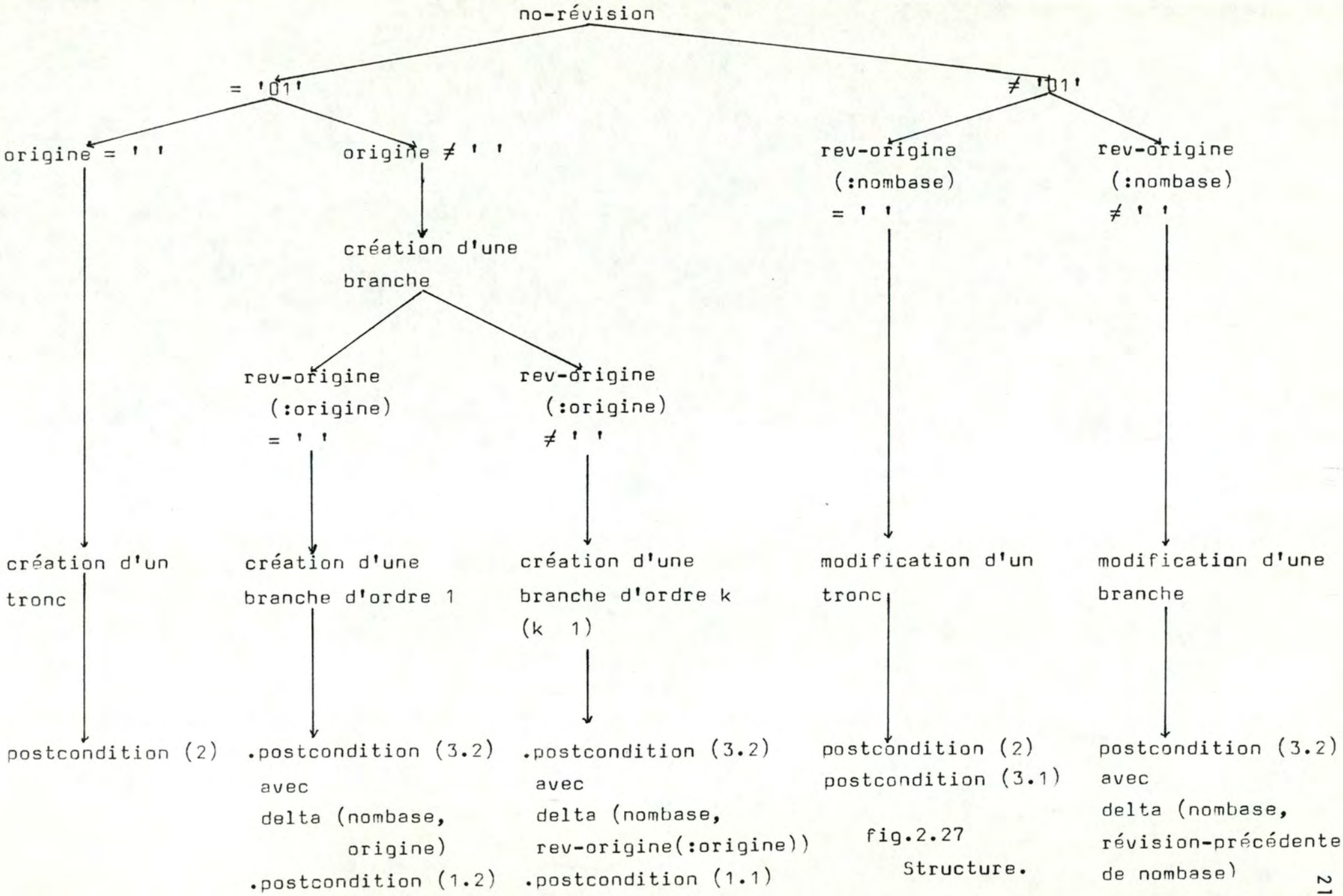


fig.2.27
Structure.

Il convient maintenant de raffiner les opérations de manière à atteindre les postconditions concernées par chacune d'entre elles.

1) Création d'un tronc.

```
COPIE (nombase, fichloc):
où la routine "copie" permet de copier physiquement le
contenu de fichloc dans nombase.
(postcondition (2))
```

2) Création d'une branche.

```
SUFFIXE (origine, nomlo cal-origine, suffixe):
LIREMAN (nomlocal-origine, ptman-origine):
(ptman-origine référence le manuel de nomlocal (:origine))
SI ptman-origine.rev-origine = ' ' ALORS
       création d'une branche d'ordre 1
SINON  création d'une branche d'ordre k;
```

3) Création d'une branche d'ordre 1.

```
RECONSTITUTION.REVISION (origine, "fichlocdelta1"):
SUFFIXE (nombase, nomlocal-base, suffixe):
CONCATNOM (nomlocal-base, "text.00", ".", fich-delta):
(no-revision = no-revision de nombase = 01
 et fich-original = "fichlocdelta1"
 et fich-destination = fichloc et fich-delta =
                       nomlocal(:nombase).text.00
    précondition de INTEGRATION-DIRECTE)
INTEGRATION-DIRECTE (fichlocdelta1, fichloc, no-revision,
                     fich-delta):
(postcondition (3.2))
ptman.rev-origine = origine:
(postcondition (1.2))
DETRUIRE fichlocdelta1:
```

4) Création d'une branche d'ordre k.

```
RECONSTITUTION-REVISION (ptman-origine.rev-origine,
                         "fichlocdelta1") ;
SUFFIXE (nombase, nomlocal-base, suffixe) :
CONCATNOM (nomlocal-base, "text.00", ".", fich-delta) :
(no-revision = no-revision de nombase = 01
 et fich-original = fichlocdelta1
 et fich-destination = fichloc et fich-delta =
    nomlocal(:nombase).text.00
    précondition de INTEGRATION-DIRECTE)
INTEGRATION-DIRECTE (fichlocdelta1, fichloc, no-revision,
                     fich-delta) :
(postcondition (3.2))
ptman.rev-origine = ptman-origine.rev-origine ;
(postcondition (1.1))
DETRUIRE fichlocdelta1 ;
```

5) Modification d'un tronc.

```
no-révision-précédente = ptman.révision-précédente :
CONCATNOM ("text", no-révision-précédente, ".", revprec) :
SUFFIXE (nombase, nomlocal-base, suffixe) :
CONCATNOM (nomlocal-base, revprec, ".", nom-revprec) :
(nom-revprec = nom de la révision précédente de nombase.
 Cette révision étant gardée en clair, on n'a pas besoin
 de la reconstituer)
COPIE (nombase, fichloc) :
(postcondition 2)
CONCATNOM (nomlocal-base, "text.00", ".", fich-delta) :
(no-révision-précédente = no de la révision n
 et fich-origine = nom-revprec = révision n
 et fich-destination = nombase = révision n+1
 et fich-delta = nomlocal (:nombase).text.00
    précondition de INTEGRATION-INVERSE)
```

```
INTEGRATION-INVERSE (nom-revprec, nombase, no-révision-
                     précédente, fich-delta) ;
(postcondition (3.1))
DETRUIRE nom-revprec ;
```

6) Modification d'une branche.

```
no-révision-précédente = ptman.révision-précédente ;
CONCATNOM ("text", no-révision-précédente, ".", revprec) ;
SUFFIXE (nombase, nomlocal-base, suffixe) :
(nom-revprec = nom de la révision précédente de nombase)
RECONSTITUTION (nom-revprec, fichlocdelta1) ;
CONCATNOM (nomlocal-base, "text.00", ".", fich-delta) ;
(no-révision = no révision de nombase = n+1
 et fich-origine = fichlocdelta1 = révision n
 et fich-destination = fichloc = révision n+1
 et fich-delta = nomlocal (:nombase).text.00
    précondition de INTEGRATION-DIRECTE)
INTEGRATION-DIRECTE (fichlocdelta1, fichloc, no-révision,
                     fich-delta) ;
(postcondition (3.2))
DETRUIRE fichlocdelta1 ;
```

L'algorithme résultant consiste en l'intégration de ces différents blocs en veillant à mettre en initialisations communes ou clôtures communes, les différents ensembles d'instructions communs aux blocs appartenant à la même alternative.

2.3.7. Reconstitution d'une révision.

Ce module est appelé par la procédure de lecture associée à la commande LIRE.

2.3.7.1. Spécification.

Ce module permettra de reconstituer une révision j d'un arbre de versions à partir d'une révision gardée en clair et de deltas. A cette fin, elle reçoit en entrée:

- un nom de version ;
- un numéro de révision ;
- un fichier de sortie ;

Le résultat de cette procédure sera le texte d'une révision dont le nom et le numéro sont contenus respectivement dans nom de version et numéro de révision. Ce résultat sera placé dans le fichier de sortie. Dans le cas où la révision n'existe plus, un message d'erreur apparaîtra.

2.3.7.2. Réalisation.

Nous avons vu au paragraphe 2.1.2. qu'il existe deux types de versions dans une arborescence: une version tronc dont la dernière révision est gardée en clair et des versions branches dont aucune révision n'est gardée en clair.
L'algorithme à mettre en oeuvre doit pouvoir régénérer dans un fichier de sortie S la $j^{ème}$ révision d'une version X d'une arborescence à partir d'une révision gardée en clair et d'un ensemble de deltas.

L'algorithme va par conséquent s'orienter vers la structure suivante:

- vérifier que la j^{ème} révision de X existe ;
- X appartient - elle à une version tronc ?
  - oui alors j est-elle la dernière révision de X ?
    - oui alors mettre le contenu de X.j dans B ;
    - non alors régénérer j et placer le résultat dans B ;
  - non alors régénérer la révision de la version tronc dont est issue X ;
    régénérer j et placer le résultat dans B ;

Il y a donc deux types de problèmes: une vérification et trois régénérations.

(1) Vérification: il existe dans la base une procédure verifnomelem qui vérifie si un objet de nom donné existe bien dans la base.
(2) Régénération: ce type de problème apparaît trois fois dans l'algorithme.

- régénérer j à partir de la dernière révision de X et de deltas (X étant la version tronc) ;
- régénérer la révision de la version tronc dont X est issue (X étant une version branche) ;
- régénérer j à partir du résultat de cette régénération et de deltas.

Commençons d'abord par le deuxième cas. Deux sous-problèmes se présentent à nouveau:

- retrouver le nom et le numéro de la révision du tronc à régénérer ;
- régénérer cette révision :

Le premier se résoud facilement car le nom et le numéro de cette révision sont disponibles dans un champ spécial du manuel appelé rev-origine (cfr.2.2.).
Le second est simple également car il suffit d'appeler récursivement la procédure que nous décrivons avec les paramètres nom de version et numéro de révision trouvés dans le manuel, et le nom d'un fichier temporaire qui devra être détruit après le traitement.

Attardons-nous maintenant aux deux autres régénérations qui semblent identiques.
Il faut régénérer une révision j à partir d'un texte qui a été soit gardé tel quel, soit régénéré, et de deltas qui sont inverses dans le cas d'une version tronc et directs dans celui d'une version branche. La question est donc de savoir quels impacts ont des deltas directs ou inverses sur la régénération d'une révision. Au §2.3.2., nous avons vu qu'un delta est composé de commandes d'édition. Tous les deltas concernant une même version sont regroupés dans un même fichier et séparés par une commande d'éditeur qui signifie: si la révision à reconstituer est égale à celle qui vient d'être régénérée par le delta précédent, ne pas appliquer les deltas ultérieurs. La différence entre un fichier de deltas directs et un fichier de deltas inverses réside simplement dans l'ajout d'un nouveau delta. Dans le cas de deltas inverses, le nouveau delta est concaténé au-dessus du fichier, et dans celui de deltas directs, la concaténation se fait au bas du fichier.
Au niveau de la régénération, cela implique que la suite des révisions reconstituées par des deltas inverses sera i,i-1,...,j+1,j (i étant la dernière révision et j la révision à reconstituer). Pour les deltas directs, la suite sera 1,2,...,j si j est la révision à reconstituer. On peut remarquer que le résultat est le même et par conséquent considérer ces deux régénérations comme identiques.
Comme chaque delta est séparé du suivant par un test sur

le numéro de révision à reconstituer (cfr.infra), la reconstitution devient assez simple et est réalisée par l'exec-com "const.ec".
Elle a en entrée quatre paramètres:

- le nom de la révision d'origine d'une version tronc ;
- le nom du fichier des deltas ;
- le numéro de révision à reconstituer ;
- le nom du fichier dans lequel la révision reconstituée se retrouvera.

Elle exécute successivement les fonctions suivantes:

- appelle d'abord l'éditeur de texte Ted ;
- place dans le buffer t1 le fichier dont le nom est contenu dans le premier paramètre :
- place dans un buffer t2 le fichier des deltas :
- fait exécuter le fichier de commandes se trouvant dans t2 sur t1 en passant comme paramètre le numéro de révision à reconstituer :
- écrit le résultat se trouvant dans le buffer 0 sur le fichier dont le nom est contenu dans le quatrième paramètre.

2.3.8. Destruction de révisions. (DESTRUCREV)

Ce module sera appelé par le module détruire de la base qui lui-même est activé par la commande détruire. Il appelle DESANT (cfr.2.3.9.) et DES2REV (cfr.2.3.10.).

2.3.8.1. Spécification.

Ce module sera utilisé lorsqu'il faudra détruire des révisions existantes d'une version

branche soit antérieures à une révision donnée, soit comprises entre deux numéros de révisions. Les bornes seront également détruites.
En entrée, il possède trois paramètres:

- nom de version ;
- numéro de révision inférieur qui sera vide dans le cas d'une destruction antérieure à un numéro de révision ;
- numéro de révision supérieur.

Le résultat de cette procédure aura des effets précisés dans les postconditions sur deux types d'objet, le manuel et l'ensemble des deltas, de la version dont le nom est contenu dans le premier paramètre.
Les cas d'erreurs possibles sont:

- err 1: la version de nom de version est une version tronc ;
- err 2: la version de nom de version n'existe pas ;
- err 3: la révision correspondant au numéro inférieur n'existe plus ;
- err 4: la révision correspondant au numéro supérieur n'existe plus.

Les postconditions sont:

err 1 implique message 1 produit
et err 2 implique message 2 produit
et err 3 implique message 3 produit
et err 4 implique message 4 produit
et ($\overline{\overline{err\ 1}}$ et $\overline{\overline{err\ 2}}$ et $\overline{\overline{err\ 3}}$ et $\overline{\overline{err\ 4}}$) implique
si numéro inférieur n'est pas vide alors
nouveau fichier des deltas = ancien fichier des deltas $\setminus$ {deltas(x,y) tq
$y \geqslant$ numéro inférieur
et $x \leqslant$ numéro supérieur}

sinon

nouveau fichier des deltas = ancien fichier \ {deltas(x,y) to x ≤ numéro supérieur}

et nouveau manuel = ancien manuel \ les numéros de révision correspondant aux révisions détruites ;

où message 1 = il est interdit de détruire des révisions de cette version ;
message 2 = cette version n'existe pas ;
message 3 = la révision correspondant au numéro inférieur n'existe pas ;
message 4 = la révision correspondant au numéro supérieur n'existe pas.

2.3.9. Destruction antérieure. (DESANT)

Ce module est appelé par DESTRUCREV.

2.3.9.1. Spécification.

Les entrées sont:

- fichier des deltas ;
- numéro de révision ;

Préconditions:

- le fichier des deltas existe et possède un format correct ;
- le delta correspondant au numéro de révision existe ;

Les sorties sont:

- fichier des deltas ;

Postcondition :

- fichier des deltas = fichier des deltas \
  {delta(x,y) tq x ⩽ numéro de révision ;}

2.3.10. Destruction entre deux révisions.

Ce module est appelé par DESTRUCREV.

2.3.10.1. Spécification.

Les entrées sont:

- fichier des deltas ;
- numéro de révision inférieur ;
- numéro de révision supérieur ;

Préconditions:

- le fichier des deltas existe et possède le format correct ;
- le delta correspondant au numéro inférieur existe ;
- le delta correspondant au numéro supérieur existe ;

Les sorties sont:

- fichier des deltas ;

Postcondition :

- fichier des deltas = fichier des deltas
  {delta(x,y) tq y ⩾ numéro inférieur
  x ⩽ numéro supérieur ;}

3. Discussion.

Cette troisième partie est consacrée aux différents aspects liés à l'intégration de la gestion des historiques et aux fonctions de mémorisation de versions. Il sera également question d'une brève discussion portant sur les contraintes d'implémentation.

Nous avons profité de l'existence de la base de programmes pour développer nos différents modules. Bien que cela puisse paraître paradoxal, nous avons constaté que le système de gestion de la base lui-même a été développé en se servant de la base elle-même. A partir du moment où le module de gestion des commandes (lire, catal ...) a été réalisé, rien n'était plus facile que d'agencer les modules ultérieurs dans la structure de familles, interfaces et versions. La base présentait donc une structure arborescente du type présenté au paragraphe 1.3.2.. Pour réaliser nos projets, nous avons travaillé au niveau de deux familles, RESCATAL et LIRE. La figure ci-dessous synthétise donc le contexte de notre travail.

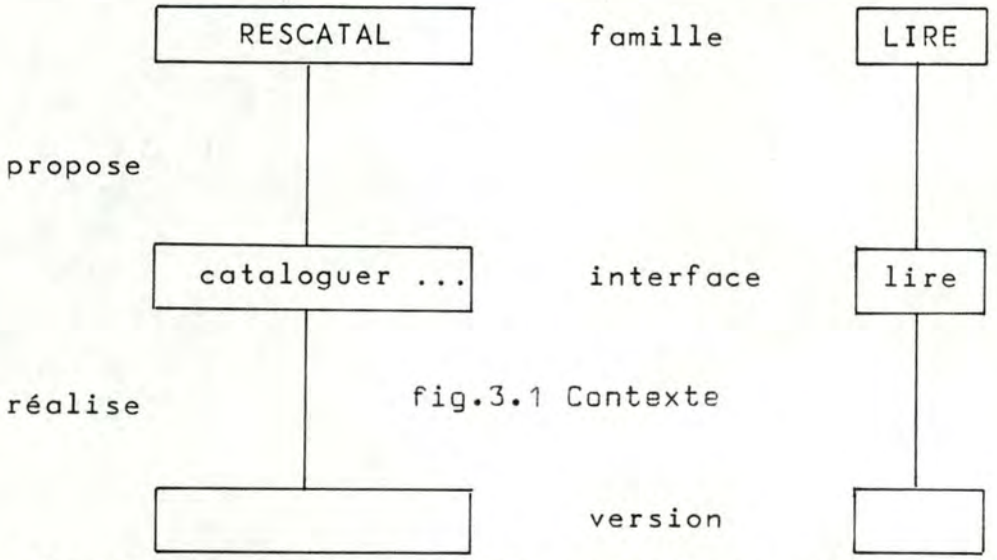


fig.3.1 Contexte

L'intégration de la gestion des historiques s'est effectuée

de la façon suivante:

- création d'une famille autonome "historique" ;
- création d'une relation de dépendance entre les familles rescatal et historique (rescatal dépend de historique) ;
- création de l'interface et d'une première version de la famille historique ;
- modification de la version de rescatal. Plutôt que de supprimer le version existante de rescatal, nous en avons créé une seconde version. Ceci nous permettait en effet de développer et tester notre nouvelle version en nous servant toujours de la version préalable.

Malheureusement, lors de l'intégration, nous avons constaté que la procédure "cataloguer" devait recevoir un paramètre supplémentaire notifiant si l'utilisateur désire commenter son catalogage. Nous avons donc créé une seconde interface présentée dans le schéma ci-dessous:

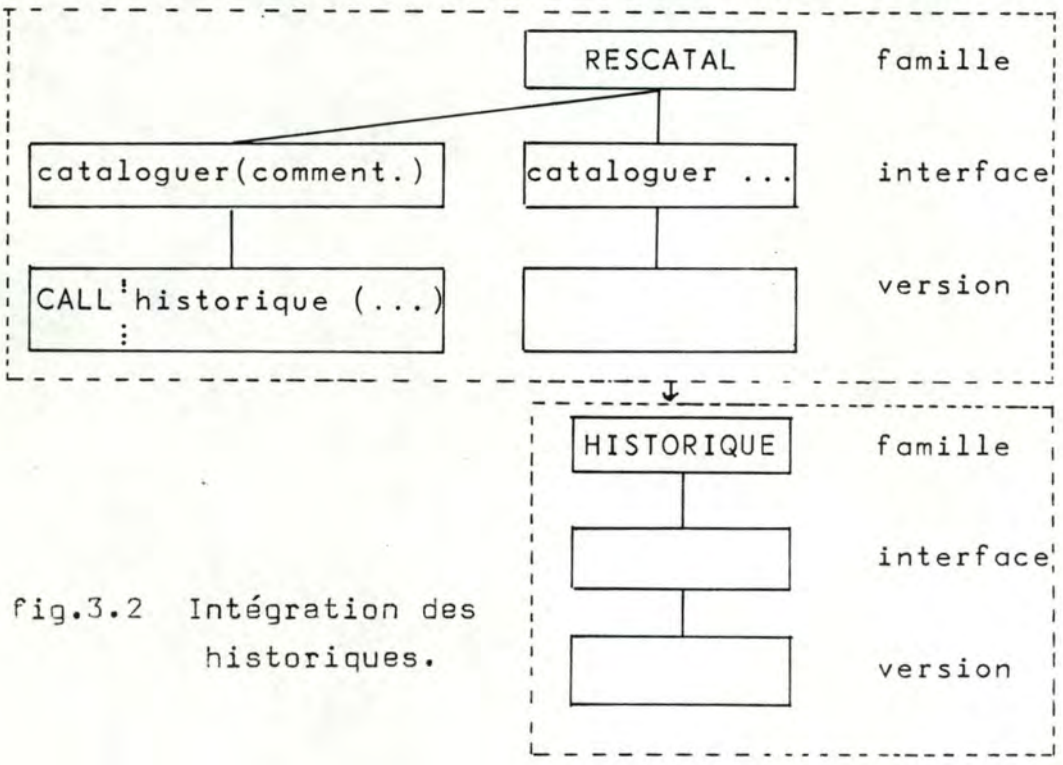


fig.3.2 Intégration des historiques.

En ce qui concerne le module LIREHIST, nous avons procédé de la même manière. Le tout est résumé dans le schéma suivant :

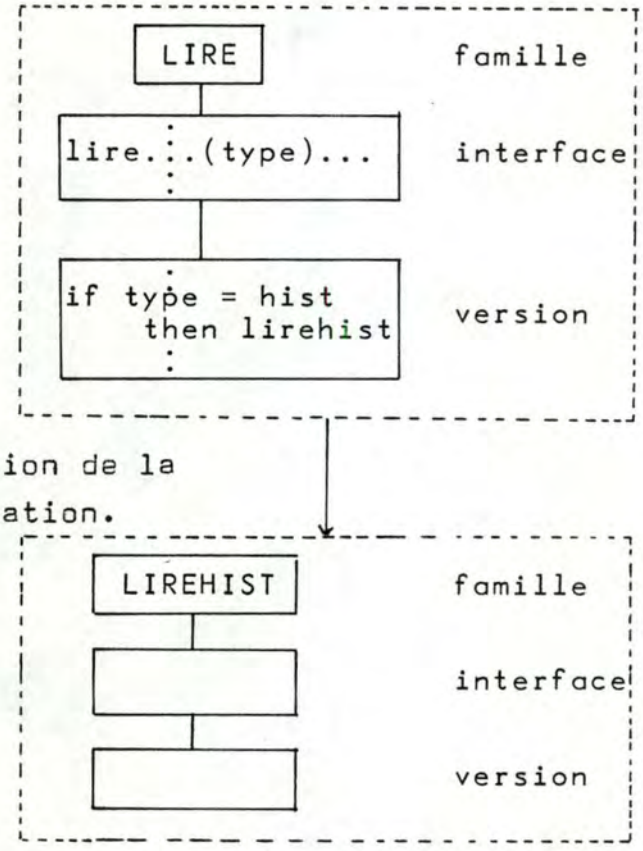


fig.3.3 Intégration de la visualisation.

Il est à noter que nous disposons à ce stade de deux versions utilisables de la base: une première version antérieure à notre développement et la même version complétée d'une gestion d'historiques.
D'autre part, les différentes exec-coms et macros Ted sont maintenues à l'extérieur de la base. Par conséquent, ces dernières ne sont pas sujettes à versions ou révisions multiples d'une façon indépendante: une nouvelle version d'une exec-com s'accompagnera nécessairement d'une nouvelle version du module qui l'utilise. Il en est de même pour l'éditeur de texte Ted qui n'est pas intégré dans la base. Dès lors tout changement de Ted pourrait avoir de sérieuses conséquences.

En ce qui concerne la gestion des deltas, notre façon de procéder est résumée dans les deux schémas suivants. On

remarque que comme pour les historiques, nous avons simplement ajouté une version dans les familles adéquates et créé deux nouvelles familles à savoir, CONSTRUIREDELTA et DELTA. On peut également constater qu'à ce stade la base existe en trois versions:

- la version primaire ;
- la version avec historiques ;
- la version avec historiques et deltas.

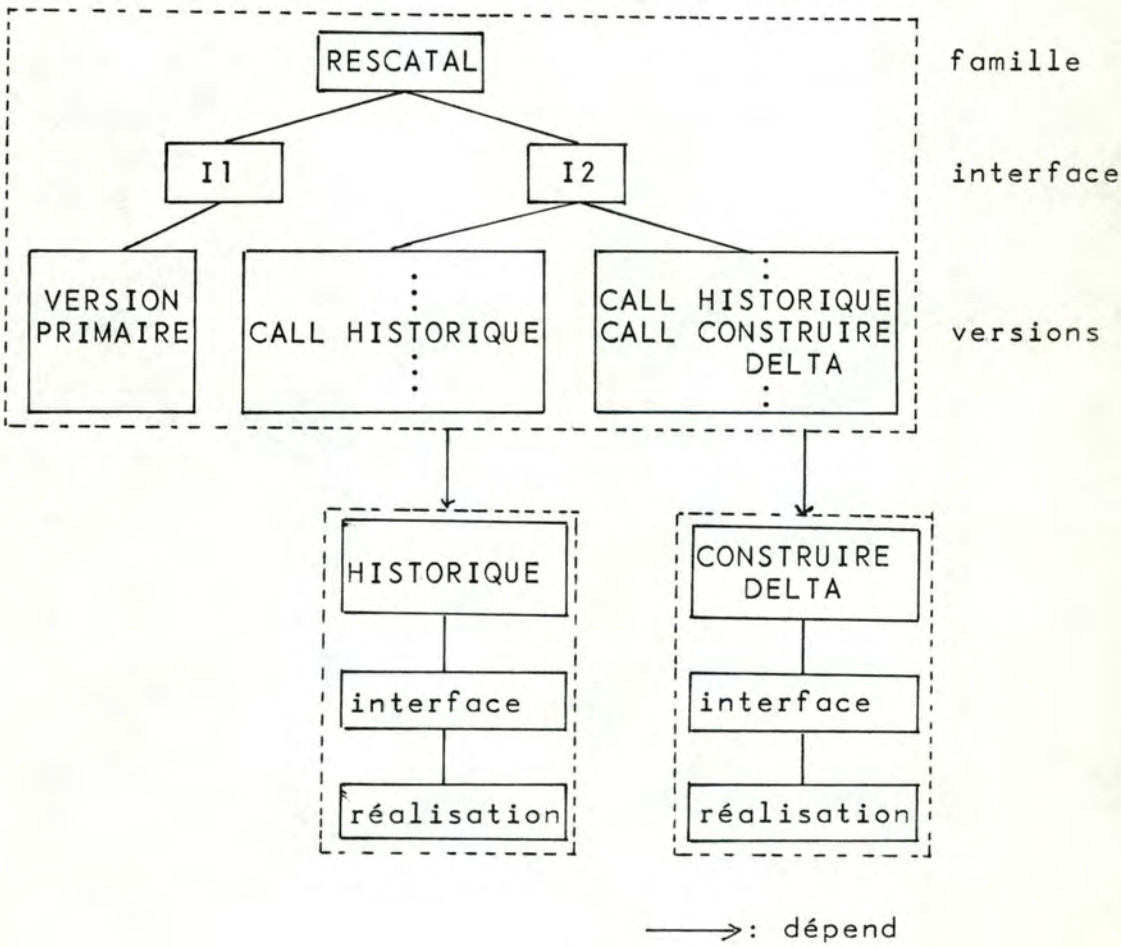
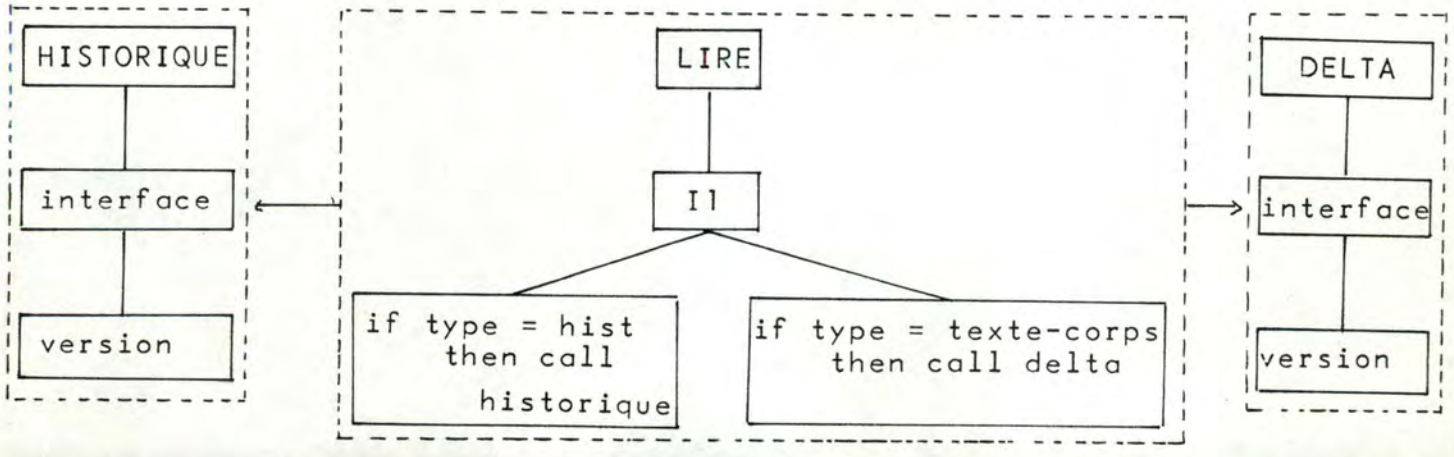


fig. 3.4 Intégration générale.



L'intégration n'a en fait posé que peu de problèmes. Cette facilité est due au développement modulaire de la base existante. On s'est aperçu de cette manière que la modularité est un moteur essentiel à l'extensibilité. L'implémentation modulaire a été facilitée par l'utilisation de la norme SOL de Pascal. Ce langage se distingue du Pascal standard par l'ajout de quelques primitives permettant d'exporter et d'importer des procédures, variables, types, fonctions ..., vers des (ou de) programmes extérieurs. Nous n'avons par conséquent pas eu de grosses contraintes de langage lors de l'implémentation. Par contre, certaines procédures Pascal se sont avérées contraignantes du point de vue du temps d'exécution. On nous a dès lors imposé d'utiliser des commandes d'éditeur partout où cela était possible. Nous avons ainsi privilégié l'efficacité à la portabilité.

Nous avons simplement une petite critique à émettre: une limite de notre travail réside en effet dans l'utilisation abusive de MULTICS à travers tous les modules, ce qui a comme effet de diminuer la portabilité. Avec le recul, il nous apparaît maintenant qu'il eût été préférable de définir une interface abstraite (Par, 77) reprenant les différentes fonctions souhaitées. Il aurait dès lors suffi de créer un module implémentant ces différentes primitives en fonction du système existant (Multics).

4. Evaluation des produits réalisés.

Ce chapitre est consacré à une discussion de notre travail. Nous aborderons, dans une première partie, la gestion des historiques en explicitant ses avantages, ses inconvénients et ses limites. Une seconde partie sera consacrée à la gestion des deltas.

4.1. Gestion des historiques.

Comme nous l'avons dit au chapitre précédent, nous avons développé nos modules en utilisant la base existante. Lorsque la gestion des historiques a été terminée, nous avons par conséquent utilisé la version de la base incluant cette gestion pour développer notre gestion de fichiers différentiels. Nous avons dès lors eu la possibilité d'apprécier les avantages, les inconvénients et les limites du travail réalisé.

4.1.1. Avantages.

Les historiques nous ont d'abord permis de constater aisément où nous en étions dans le catalogage de nos différents modules. Ceux-ci étaient en effet développés dans des directories personnelles et il importait souvent de savoir si la version se trouvant dans la base était plus récente ou plus ancienne que celle se trouvant dans la directory de travail. La date et l'heure du catalogage enregistrées dans l'historique nous permettaient de trancher ce problème. Cela est particulièrement important lorsque plusieurs personnes sont susceptibles de cataloguer une même version.

D'autre part, un historique pouvait nous dire si un code objet

quelconque se trouvant dans la base, correspondait bien au code source associé. En effet, lors de la commande "catal" du texte d'une réalisation, l'utilisateur peut prendre l'option de cataloguer le code objet associé. Cela s'opère par la commande: catal X.text.y -co .
La gestion des historiques enregistre alors un élément d'historique pour le source, avant d'enregistrer un autre élément pour l'objet. Il suffit dès lors de visualiser l'historique pour vérifier la correspondance des deux codes. Rappelons que Make (Fel, 79) offre la même possibilité. Chaque objet est en effet estampillé par la date de leur dernière modification et il est dès lors aisé de voir si un code objet est antérieur ou postérieur à la réalisation associée.

Un autre avantage de la gestion des historiques réside dans le commentaire. C'est d'ailleurs cette information que nous regardions en premier lieu lorsque nous consultions un historique. Il nous permettait en effet de connaître la raison d'un catalogage et de pouvoir nous remémorer rapidement si un code objet avait été recatalogué suite aux modifications que nous y avions apportées. L'utilisateur peut y inclure toute information qu'il juge pertinente et ne doit obéir à aucune structure prédéfinie. Il est d'autre part facultatif de manière à accélérer les catalogages moins importants.

D'autre part, la visualisation d'un historique a été conçue d'une manière suffisamment souple afin de favoriser une utilisation plus pratique. C'est ainsi que l'on peut consulter les éléments d'un historique enregistrés entre deux dates précisées par l'utilisateur, ou encore ne visualiser que les éléments d'un type particulier (man, text, doc, co).
Les dates peuvent être fournies sous plusieurs formats et le caractère "*" peut remplacer les dates initiales ou finales.
Remarquons également que le procédé de visualisation reste identique quel que soit l'historique consulté. L'utilisateur

procédera donc de la même manière pour l'historique d'une famille, d'une interface ou d'une réalisation. Il pourra aussi choisir de visualiser son historique à l'écran ou d'en constituer un fichier de travail qu'il pourra plus tard sortir à l'imprimante.
Rappelons d'autre part que les historiques de versions différentes ne sont pas nécessairement indépendants. En effet, si une révision d'une version A est modifiée afin de créer la première révision d'une version B, les historiques associés à A et B gardent la trace de la jonction. Nous pouvons ainsi rappeler à l'utilisateur quel est l'ensemble des versions ascendantes à ou descendantes d'une version déterminée. Cela peut s'avérer utile lorsque les arborescences de versions deviennent importantes.

Nous avons finalement imposé la stabilité des historiques en empêchant leur modification. Tout catalogage d'un historique est à cette fin refusé.

Quant à l'insertion d'informations dans le texte des réalisations, elle nous a permis de toujours savoir avec exactitude quelle version de réalisation nous étions en train de modifier. Le nom de l'objet et les attributs étaient insérés à cet effet. Nous pouvions également connaître l'auteur et la date de la dernière modification, et un journal nous permettait d'apprécier l'évolution chronologique de notre travail.

En ce qui concerne la réalisation, nous avons surtout essayé de cacher ce qui risquait de changer dans des modules particuliers. C'est ainsi que la structure d'un fichier historique est encapsulée dans un module spécialement destiné à la gérer et que toute modification de cette structure pourra dès lors s'opérer sans inconvénient.
L'implémentation par exec-com et macros Ted a conduit à une utilisation relativement rapide et l'utilisateur n'a pas

trop à souffrir des caprices du CPU.

4.1.2. Limites.

La première limite de la gestion des historiques provient du fait que nous ne gardons la trace que des modifications des différents objets de la base. Ceci faisait en effet au départ partie de nos objectifs et délimitait par conséquant la frontière de notre travail. Nous nous sommes néanmoins rendus compte qu'il eût été profitable de conserver également la trace de la destruction des objets afin de savoir qui était l'auteur de l'opération, quand et pourquoi elle s'était déroulée.

La non modifiabilité du commentaire d'un élément d'historique constitue une seconde limite. Nous refusons en effet la modification de tout élément d'historique et le commentaire respecte par conséquent la règle. Cette limite était évidemment décidée à priori mais nous nous sommes aperçus à posteriori qu'elle constituait en fait un inconvénient. Plusieurs utilisateurs nous ont en effet fait la remarque suite à des fautes d'orthographe, ou à des erreurs ou oublis dans leurs commentaires. Bien que conscients de cette lacune, nous leur avons répondu que l'historique ne constituait qu'une trace et que ce qu'on y enregistrait était largement suffisant pour se souvenir à tout moment des modifications effectuées. D'autre part, l'expérience nous a montré qu'on se souvient parfois plus aisément d'un commentaire erroné que d'un commentaire sans erreur.

Une dernière limite réside dans le fait que les modifications effectuées sur des objets de la base ne sont pas enregistrées dans le cadre de la gestion des historiques. Certains auraient en effet voulu connaître grâce à cet outil, les changements entre deux révisions successives, et d'une

façon générale, ils auraient voulu savoir quelles étaient les transformations opérées sur les différents objets de la base. Notons que pour des objets tels que les manuels, les transformations peuvent aisément être inclues par l'utilisateur dans son commentaire. Pour des objets tels que le texte des réalisations, l'objection paraît plus pertinente: nous remarquerons cependant que la gestion des deltas peut nous permettre de répondre à ce type de questions.

4.1.3. Inconvénients.

Comme nous l'avons remarqué à plusieurs reprises,lors de l'explicitation technique de notre travail, nous avons été contraints, pour des raisons de performance, d'utiliser des macros Ted et des exec-coms. Les unes et les autres représentent un facteur non négligeable compromettant la portabilité. Cela est d'autant plus préjudiciable que la base ADELE a depuis lors été vendue et est destinée à être portée sur des systèmes différents de celui sur lequel elle a été réalisée.

Un second inconvénient, intrinsèque à la gestion des historiques, provient de l'enregistrement du nom de l'objet dans le texte des réalisations. Lors de la modification du texte d'une réalisation, ce nom est utilisé afin de connaître la dénomination de l'objet modifié et de vérifier si la modification donne naissance à une nouvelle version. La modification de ce nom, par inadvertance ou volontairement, dans le chef de l'utilisateur, pourra causer des erreurs dans l'enregistrement des historiques. Des mesures de prévention limitent toutefois dans une large mesure les changements involontaires. Nous contrôlons en effet si le nom de l'objet figurant éventuellement dans la première ligne du texte d'une réalisation est un nom connu de texte de réalisation dans la base.

4.2. Gestion des deltas.

Rappelons la stratégie que nous avons adoptée pour la gestion des deltas. La dernière révision de la version centrale, encore appelée version tronc, est gardée en clair, tandis que des deltas inverses sont calculés pour les révisions antérieures. Pour les versions branches, des deltas directs sont calculés, et le premier delta calculé est celui qui unit la première révision de la branche à la révision du tronc dont elle provient.
Cette technique peut être résumée de la manière suivante:

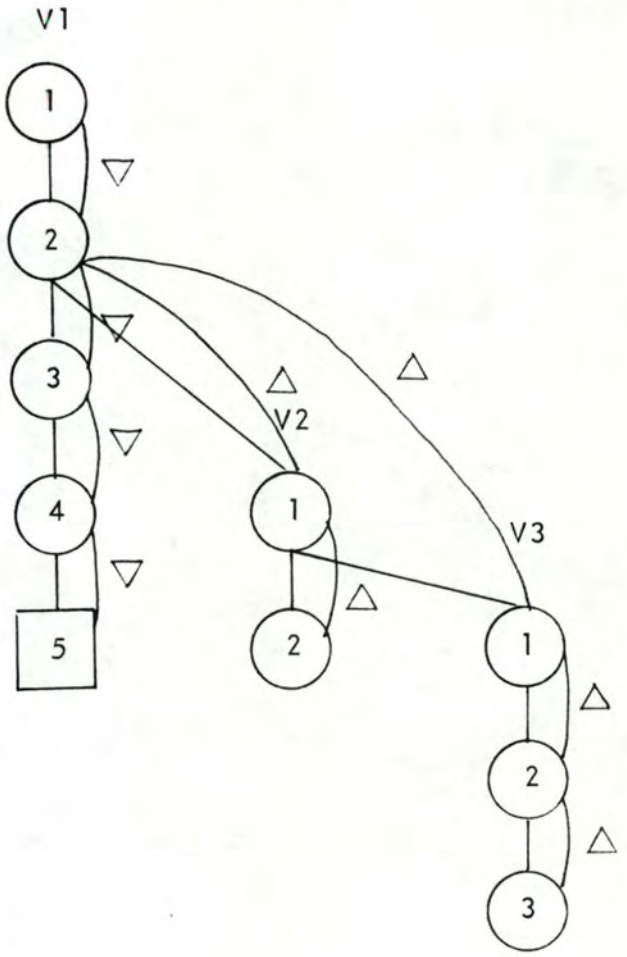


fig.4.1 Rappel de la solution adoptée.

4.2.1. Avantages.

Un premier avantage de cette technique réside dans le fait que nous ne gardons qu'une seule révision par arborescence de versions. Toutes les autres révisions peuvent être générées par l'application des deltas appropriés. L'avantage provient évidemment du fait qu'un delta est de taille largement inférieure à celle du texte d'une réalisation.

D'autre part, lorsque l'utilisateur crée une nouvelle version à partir d'une version existante (par exemple, V2.1 à partir de V1.2), il peut choisir de créer une version indépendante de l'arborescence, ou au contraire, il peut décider de l'intégrer à l'arborescence en concrétisant son existence par le calcul d'un nouveau delta. Cette souplesse laisse à l'utilisateur certains degrés de liberté qui lui permettent de créer une version indépendante lorsqu'il considère que les différences entre ses deux versions sont trop importantes.

Un troisième avantage peut être trouvé dans la transparence des deltas vis-à-vis de l'utilisateur. Celui-ci a en effet l'impression que toutes les révisions sont gardées en clair, et il ne possède aucun moyen de deviner la stratégie de mémorisation.

La technique de gestion du fichier des deltas a, d'autre part, permis une reconstitution rapide. Le texte intégral de la réalisation, le fichier des deltas et le numéro de révision sont les seuls ingrédients nécessaires à la reconstitution de toute révision d'une branche particulière. La regénération d'une révision appartenant à une version branche d'ordre supérieur à 1 est accélérée par le fait que toutes les versions sont directement rattachées à la version tronc. Quant à la constitution des deltas, elle n'augmente le temps

du catalogage que d'une façon limitée et inespérée.

En ce qui concerne la réalisation de cet outil, elle a été conçue d'une façon suffisamment modulaire que pour pouvoir changer de stratégie relativement aisément. Nous avons d'ailleurs d'abord développé une technique où la dernière révision de toute version était gardée en clair, et nous avons changé de stratégie sans aucune difficulté. De la même manière, nous pouvons passer à une nouvelle version qui supprimerait les liens directs des branches au tronc des arborescences.

4.2.2. Limites.

La frontière de notre travail était définie de manière telle que nous n'avons rien prévu si l'enregistrement des différences entre deux révisions est de taille supérieure à la révision épargnée. Notons que cela est possible suite à l'inclusion dans les deltas d'une série de caractères de contrôle de l'éditeur. Il serait cependant facile de calculer la taille des deux fichiers et d'avertir l'utilisateur du rapport des grandeurs. D'autre part, il serait agréable que le travail soit couplé à un outil de statistique pour que l'utilisateur puisse aisément estimer sa rentabilité. On pourrait, par exemple, mémoriser la place gagnée par l'utilisation des différences lors des catalogages. Cela permettrait de calculer, après un certain laps de temps, des valeurs caractéristiques telles que moyenne et variance. On aurait ainsi une preuve mathématique de l'efficacité ou de l'inutilité de l'outil.

4.2.3. Inconvénients.

Le manque de portabilité, pour les mêmes raisons que la gestion des historiques, constitue évidemment

un inconvénient majeur.
D'autre part, le nom de l'objet inclus dans la première ligne des réalisations présente le même type d'inconvénient que celui explicité en 4.1.3. . Si l'utilisateur le transforme de quelque façon que ce soit, le delta risque d'être calculé entre deux révisions qui n'ont aucune relation entre elles. Cela ne risque pas de provoquer des pertes de révisions mais bien d'augmenter la taille des deltas. C'est dans ce type de situation que le calcul de la taille des différences s'avérerait indispensable.

4.2.4. Avantages ou inconvénients ?

Le fait que les deltas ne soient pas fusionnés ou condensés consitue-t-il un avantage ou un inconvénient ?
Supposons que les différences entre les révisions 3 et 4 d'une version soient l'inverse de celles entre les révisions 4 et 5. Il est certain que si les deltas sont conservés tels quels, sans fusion qui conduirait à un delta nul entre les révisions 3 et 5, la reconstitution de la révision 3 va constituer un aller-retour causant une perte de temps injustifiable. D'autre part, le fait de conserver le delta entre les révisions 3 et 4 constitue une perte de place. Mais dans combien de cas, cette situation inhabituelle va-t-elle se produire ? Il est bien certain qu'en ne fusionnant pas les deltas successifs, on perd de la place et du temps à la reconstitution. Mais il n'en est pas moins vrai qu'une telle condensation risque de grever dans une large mesure le temps de constitution d'un delta.

Quant au fait que toutes les versions sont liées au tronc, certains diront que c'est un avantage, d'autres un inconvénient, suivant qu'ils se placent au niveau de la rapidité de reconstitution ou au niveau du gain de place.

En ce qui concerne la destruction d'un ensemble de révisions, elle présente l'avantage qu'elle est entièrement gérée par la base avec toutes les difficultés qu'elle occasionne. Des deltas doivent en effet parfois être recalculés. Ce recalcul comporte l'inconvénient que des deltas sont parfois générés entre des révisions étrangères. Si on supprime V3.1 et V3.2 dans l'exemple présenté en 4.2., il faudra recalculer un delta entre V1.2 et V3.3 qui sont deux révisions relativement éloignées.

5. Extension: la mémorisation des réalisations sous une forme arborescente.

Cet ultime chapitre est consacré à l'étude de la conservation des programmes sous une forme intermédiaire plutôt que sous une forme textuelle.
Il y est également question des impacts qu'une telle conservation aurait sur la gestion des deltas: est-elle possible, profitable ou néfaste ?

5.1. Introduction.

La plupart des éditeurs de texte se limitent à manipuler des lignes composées de caractères. Un programme étant constitué d'unités syntaxiques conformes à une grammaire rigoureusement définie, ces notions de lignes ne sont pas toujours des plus appropriées pour manipuler cet objet. Bien qu'elles paraissent pratiques et que leur manipulation entraîne des modifications rapides, il n'en est pas moins vrai qu'un programmeur utilise intuitivement le concept d'unité syntaxique lorsqu'il modifie son programme. Il suffit pour s'en convaincre d'observer quelqu'un en train de tracer des lignes verticales sur un listing pour en faire émerger la structure.

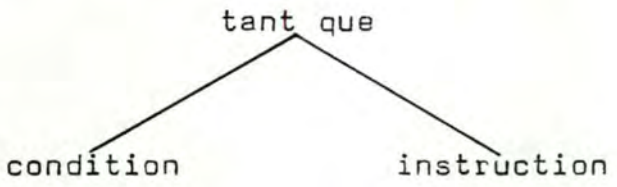
Cette manipulation intuitive est actuellement de plus en plus concrétisée par l'apparition des éditeurs syntaxiques. Citons entre autres Mentor (Don, 79), CPS (Tei, 80), Adèle (ROU, 84a). Cette apparition ne résulte évidemment pas du besoin de concrétiser une intuition, mais de celui de réduire le cycle d'écriture d'un programme. En effet, un tel éditeur connaît la syntaxte du texte qu'il reçoit, et peut dès lors effectuer une série de contrôles ou éviter des erreurs triviales telles que l'omission d'un end, d'un point-virgule ou d'une parenthèse, telles que des fautes d'orthographe de mots-clés, ... Pour donner un autre exemple de la puissance des édi-

teurs syntaxiques par rapport aux éditeurs de texte, nous remarquerons que la substitution de toutes les occurences "gin" par "rhum" effectuée par celui-ci ne remplace pas le mot-clé "begin" par "berhum", alors que c'est le cas pour les éditeurs conventionnels.

De par leur connaissance de la syntaxe, les éditeurs syntaxiques transforment, d'autre part, facilement le texte entré par l'utilisateur en une représentation arborescente qui en reflète exactement la structure. L'instruction Pascal "tant que" écrite textuellement

WHILE (condition) DO (instructions)

sera transformée en l'arborescence



Ce reflet de la structure apparaît particulièrement intéressant pour des outils tels que les interprèteurs-metteurs au point ou les générateurs de code. Il nécessite la production de décompilateurs, transformant la structure arborescente en une structure textuelle, et d'introducteurs possédant la fonction inverse.
D'autre part, le reflet de la structure fait dire à certains que cette représentation arborescente, encore appelée représentation interne (RI), est plus riche qu'une représentation textuelle qui ne fait ressortir la structure que dans une moindre mesure par l'intermédiaire d'artifices tels que l'indentation. D'autres argumentent qu'elle n'est ni plus riche ni plus pauvre puisqu'elle renferme exactement la même information sous une autre forme. La richesse d'une RI réside, selon nous dans le fait qu'elle permet une utilisation plus aisée par les outils décrits au début de ce travail, et que bien qu'elle renferme la même information qu'une représen-

tation textuelle, celle-ci est plus facilement extraite. C'est précisément cette richesse qui incite la plupart des environnements de programmation actuels à utiliser cette représentation comme axe principal. Nous avons constaté au début de ce mémoire qu'Adèle ne faisait pas exception, mis à part un outil, la base de programmes. Ceci est dû à une désynchronisation dans le développement des outils puisqu'au départ la base était conçue pour conserver ses réalisations sous une forme interne.

Le paragraphe suivant est destiné à l'étude des répercussions que ce type de mémorisation aurait eu sur la gestion des deltas.

5.2. Impacts sur la gestion des deltas.

5.2.1. Concepts.

La gestion des deltas, telle qu'elle est explicitée dans ce mémoire, utilise la notion de ligne comme atome de modification. Cela signifie que si un seul caractère est modifié dans une ligne, le système considère que toute la ligne est modifiée. Dans le cadre d'une représentation interne, l'atome de modification serait l'unité syntaxique.
D'autre part, si notre travail était couplé à un éditeur de texte pour régénérer le texte des révisions, il serait très facile de le coupler à un éditeur syntaxique pour reconstituer des arborescences. Remarquons ici que la notion d'intégration dans les ateliers logiciels prend tout son sens. La base perd son indépendance et devient inéluctablement liée ou intégrée à l'atelier par l'intermédiaire de la structure de données (RI) et de l'éditeur syntaxique.
Quant aux autres concepts plus globaux tels que delta direct, delta inverse, version branche, version tronc, lien direct entre branche et tronc ..., ils restent valides mais sont

appliqués à des unités d'information à sémantique plus riche.
Examinons maintenant les avatages et inconvénients que ces nouveaux concepts occasionnent.

5.2.2. Avantages.

Lorsqu'on considère un programme comme une suite de lignes, il est bien clair qu'un éditeur de texte fait la différence entre deux lignes comprenant exactement les mêmes caractères ($\neq\emptyset$) mais précédées ou suivies par des nombres inégaux de blancs.
Ainsi, le texte

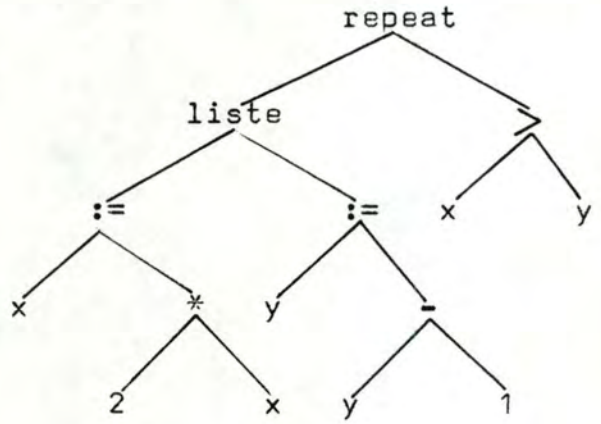
```
repeat
x:=2*x ;
y:=y-1 ;
until x>y ;
```

est totalement différent du texte

```
    repeat
       x:=2*x ;
       y:=y-1 ;
    until x>y ;
```

aux yeux d'un éditeur de texte. Cela signifie que si l'utilisateur catalogue une révision non indentée et qu'il indente la révision suivante avant de la cataloguer, le système va calculer un delta contenant toute la révision, ce qui constituera une perte de place suite aux caractères de contrôle d'édition introduits dans les différences.
En ce qui concerne une représentation arborescente, cette anomalie n'est pas apparente. En effet, les deux textes présentés ci-dessus sont transformés en une structure iden-

tique du type:



Au noeud "repeat" est associé un schéma de décompilation qui permettra de transformer cette structure arborescente en la deuxième forme textuelle présentée.

D'autre part, si l'utilisateur change la ligne "until x>y" en "until y>x", cela ne pourrait provoquer aucune génération de différence. Le module de comparaison d'arbres pourrait en effet évaluer l'équivalence sémantique de deux sous-arbres syntaxiquement différents. Remarquons que dans le cas d'un éditeur classique, la ligne entière est considérée comme étant modifiée. On peut donc constater que l'utilisation d'une RI peut occasionner des gains de place importants en ce qui concerne les deltas. Pour nous en persuader davantage, considérons l'exemple suivant:

l'utilisateur transforme l'exemple présenté ci-dessus en

```
repeat
   x:=3*x ;
   y:=y-2 ;
until x≥y ;
```

Du point de vue de l'éditeur de texte, les trois dernières lignes ont été modifiées tandis que du point de vue de l'édi-

teur syntaxique seuls un noeud et deux feuilles respectivement associés aux valeurs >, 1 et 2 sont transformés.

En ce qui concerne l'efficacité de la reconstitution des arbres, notons qu'un gestionnaire de RI possède un arsenal de primitives telles que, le remplacement d'un sous-arbre par un autre, l'insertion d'un noeud avant ou après un autre noeud, la recherche du père, du ou des fils, du ou des frères d'un noeud particulier .... Les différentes commandes de la RI Adèle sont résumées dans (Rou, 84b).

5.2.3. Inconvénients.

Il arrive, et il est même à conseiller, qu'un programme soit truffé de commentaires destinés à faciliter sa compréhension. Du point de vue d'une représentation arborescente, ceux-ci sont généralement assimilés à des feuilles. Il en résulte que la modification d'un seul caractère dans un commentaire provoque l'introduction dans un delta de la totalité de la feuille correspondante. Si le commentaire est composé de plusieurs lignes, il est dès lors évident qu'une comparaison basée sur la ligne est plus bénéfique. Notons toutefois que certains éditeurs évolués, comme Mentor, considèrent un commentaire comme une annotation qui s'associe à un noeud. Celle-ci peut être structurée dans un formalisme défini par l'utilisateur, différent de celui du langage. Rien n'empêche à l'usager d'imposer que ses commentaires soit structurés en chapitre, section, paragraphe et ligne. Cela permet d'annuler l'inconvénient cité.

Supposons d'autre part que l'utilisateur désire visualiser un delta afin de se souvenir des modifications opérées sur une réalisation. Il est évidemment habitué à lire des lignes entières. Dans l'exemple présenté précédemment, il se sou-

viendra plus aisément de sa modification s'il voit la ligne "until x $\geqslant$ y" plutôt que l'unité " $\geqslant$".

Nous pensons néanmoins que ces deux inconvénients sont largement compensés par les avantages.

CONCLUSION.

A notre arrivée à Grenoble, nous étions quelque peu dépaysés. En effet, nous devions travailler dans un contexte peu familier, celui d'un atelier logiciel intégré contenant des outils tels que décompilateur, introducteur, médiateur, compositeur, gestionnaire de RI ....
Etant surtout habitués à gérer des clients et des fournisseurs, tout cela nous paraissait obscur et plutôt énigmatique. Nous nous réjouissons d'ailleurs de l'initiative de l'Institut d'orienter les étudiants vers des domaines précis. Notre intégration a toutefois été facilitée par la disponibilité de tous les membres de l'équipe Adèle qui nous ont initiés à ces nouveaux concepts.

Après notre intégration, nous avons directement conçu et réalisé les deux outils qui font l'objet de ce mémoire.
Le premier, la gestion des historiques, nous a permis de prendre un premier contact avec la base Adèle. Etant donné que nous avons utilisé cet outil après l'avoir réalisé, nous avons pu constater qu'il était d'une réelle utilité pour la base. Ses différents avantages, inconvénients et limites ont été largement développés au chapitre 4 concernant l'évaluation critique.
Le second, la gestion des deltas, nous a plus profondément motivés. Nous en connaissions en effet a priori l'utilité. Nous nous sommes surtout inspirés de RCS, un système de contrôle de révisions développé autour du système UNIX. Les différences essentielles entre ce dernier et notre travail se résument à la technique de mémorisation des deltas et aux liens directs qui unissent toutes les versions branches à la version tronc. Nous avons privilégié la rapidité de reconstitution des révisions au volume des deltas, ceux-ci restant toutefois de taille largement inférieure à celle

des textes intégraux.

Quant à une base de type Adèle, elle nous semble indispensable dans tout environnement gérant des versions multiples de programmes. Nous avons pu constater avec quelle aisance et quelle rapidité différentes configurations pouvaient être construites. Précisons d'ailleurs qu'elle a été vendue, dans le courant de l'année 84, à la SFENA (Sociéte Française d'Etudes de Navigation Aérienne) et au CICG (Centre Informatique de Grenoble).

BIBLIOGRAPHIE.

(Boe, 82)
B.W. Boehm
"Les facteurs du coût du logiciel"
TSI, Vol. 1, no. 1, 1982, pp. 5-24

(Don, 79)
Donzeau-Gouge V., Huet G., Khan G., Lang B.
"Programming environments based on structured editors:
the MENTOR experience"

(Dro, 82)
E. de Drouas, J.M. Nerson
"Les ateliers logiciels intégrés: développements français actuels"
TSI, Vol. 1(3), 1982, pp. 211-232

(Est, 84)
Estublier J.
"Preliminary experience with a Configuration Control
System for Modular Program"
Software Engineering Symposium on Practical Software
Development Environment, Avril 1984

(Fel, 79)
S.I. Feldman
"Make: a program for maintaining software"
Software practice and experience, Vol. 9, 1979, pp. 255-265

(Fin, 82)
Finger U. et al
"Description générale de la SM90"
Note technique NT/PAA/OGE/SML/703, CNET Lannion, mai 1982

(Gho, 84) (Adèle)
Ghoul S.
"Base de données et gestion de configurations dans
un atelier de génie logiciel"
Thèse de 3ème cycle, Grenoble, 1984

(Her, 84) (Adèle)
Herrmann M.
"Interface usager-application dans un atelier de génie logiciel"
Thèse de 3ème cycle, Grenoble, 1984

(Kra, 82)
Krakowiak S.
"Systèmes intégrés de production de logiciel:
concepts et réalisations"
TSI, Vol. 1(3), 1982, pp. 187-200

(Lam, 82)
A. van Lamsweerde
"Les outils d'aide au dévéloppement de logiciels: un
aperçu des tendances actuelles"
XVème journées internationales de l'Informatique et de
l'Automatisation, Paris 16-18 juin 1982

(Len, 84) (Adèle)
Lenne C.
"Interprétation et mise au point de programmes dans
un atelier de génie logiciel"
Thèse de cycle, Grenoble, 1984

(Mic, 79)
JG. Mitchell, W. Maybury, R. Sweet
MESA langage manual
Technical Report CSL-79-3, XEROX PARC, Avril 1979

(Mul, 1)
Multics Sort/ erged Manual order no. AW32

(Mul, 2)
Multics user's guide order no. AG92-04 pp. 3415-3419

(Naf, 79)
Naffah N.
"Exemple d'un poste de travail bureautique à interface
universelle"
T.R. MEV.2.503 Projet pilote KAYAK, INRIA, juin 1979

(Par, 77)
Parnas D.L.
"Use of Abstract Interfaces in the Development of Software for Embedded Computer System"
NRL Report no. 8047, juin 1977

(Par, 81)
Britton K.H., Parker R.A., Parnas D.L.
"A procedure for designing abstract interfaces for device interface modules"

(Par, 81a)
Britton K.H., Parnas D.L.
"A-7E Software Module Guide"
NRL Report 4702, décembre 1981

(Roc, 75)
Rochkind M.J.
"The Source Code Control System"
Bell Laboratories, Murray Hill, N.J.07974, août 1975

(Rou, 84) (Adèle)
Rouzaud Y.
"Représentation et manipulation de programmes dans un atelier de génie logiciel"
Thèse de 3ème cycle, Grenoble, 1984

(Rou, 84a) (Adèle)
Rouzaud Y.
"Représentation et manipulation de programmes dans un atelier de génie logiciel"
"L'éditeur d'Adèle" pp. 41-45
Thèse de 3ème cycle, Grenoble, 1984

(Rou, 84b) (Adèle)
Rouzaud Y.
"Représentation et manipulation de programmes dans un atelier de génie logiciel"
"Opérations sur la RI" pp. 72-78
Thèse de 3ème cycle, Grenoble, 1984

(San, 84) (Adèle)
Santana M.
"Un système de production automatique de générateurs de code"
Thèse de 3ème cycle , Grenoble, 1984

(Tei, 81)
T. Teiltelbaum, T. Reps
"The Cornell Program Synthetizer: a syntax directed programming environment"
Communication of the ACM, Vol. 24 no.9, septembre 1981, pp. 563-573

(Tho, 82)
Thomson
"Micromega-32 guide utilisateur"
Thomson-CSF Département Informatique de Bureau
Paris, 1982

(Tic, 82)
WF. Tichy
"Design, Implementation, and Evaluation of a Revision Control System"
Proc. 6th International Conference on Software Engineering, Japon, 1982

(Wir, 82)
"Programming in Modula2"
Tringer-Verlag, 1983

ANNEXE A.

Grammaire BNF de la désignation des objets.

En utilisant la notation BNF (Backus-Naur Form), on peut définir la désignation des objets de la manière suivante:

<nom de famille> ::= <nom local de famille> |
<nom global de famille>
<nom global de famille>::=><nom local de famille>
<nom local de famille>::=<ident>(><ident>)*
<ident> ::= caractère(caractère)^{n} $n \leqslant 14$

<nom d'interface>::=<nom global d'interface> |
<nom local d'interface>
<nom local d'interface>::= -<ident> (<)* (>)*<nom local de famille>-<ident>
<nom global d'interface>::=<nom global de famille>-<ident>

<nom de corps>::= <nom local de corps> |
<nom global de corps>
<nom local de corps>::=<nom local d'interface>-<ident>
<nom global de corps>::=<nom global d'interface>-<ident>

Les différents types d'objets associés à ces familles, interfaces et corps se désignent en concaténant le nom du type précédé d'un point au nom de l'objet. Un manuel s'appellera

<nom de famille>.man
<nom d'interface>.man
<nom de corps>.man

suivant qu'il est respectivement associé à une famille, une interface ou une réalisation.

ANNEXE B.

Présentation des manuels.

B.1. Contenu des manuels.

B.1.1. Manuel de familles.

manuel (nom local de famille)

use
- informations en consultation uniquement
- état
  - nom complet :
  - type :
  - no universel :
  - niveau :
  - vide :
  - modifié :
  - modifdate :
  - modifusager :
  - ls incohérent :
  - ls périmé :
  - ls verrouillé :
  - ls incomplet :
- relation
  - liste des accès inverses :
  - liste des familles externes :
  - liste des familles filles :
  - liste des interfaces associées :

B.1.2. Manuel d'interfaces.

manuel (nom local d'interface)

incl (nom des interfaces incluses)

- informations en consultation uniquement
- état
  - nom complet :
  - niveau :
  - vide :
  - modifdate :
  - modifusager :
  - environ :
  - incohérent :
  - verrouillé :

- relations
  - liste-incl-inverse :
  - liste-environs-inclus :
  - liste-environs-incluants :
  - liste-dépendance-inverse :
  - liste-corps :
  - liste-realcomp :

B.1.3. Manuel de corps.

manuel (nom local de corps)

sel (contraintes de sélection)
- selimp
- selcond
- seldef
- selnon

attr (attributs)

util (utilisé : oui ou non)

- informations en consultation uniquement
- état
  - nom complet :
  - type :
  - nom universel :
  - niveau :
  - vide :
  - modifié :
  - modifdate :
  - modifusager :
  - ls incohérent :
  - ls perimé :
  - ls verrouillé :
  - ls incomplet :

- relations
  - existence de fichiers :
  - liste de dépendances :
  - liste des réalisations englobantes :
  - liste des codes objets associés :
  - numéros des révisions existantes :
  - liste des contraintes héritées :

Nous donnons ci-après un exemple de manuel de famille et un exemple de manuel de corps.

a) manuel de la famille b:


```
manuel  b
        def

        use

end
informations en consultation uniquement:

        nom complet     :  >ADL>accesbase>b.man
        type            :  de famille
        nom universel   : nr4
        niveau          :              3
        vide            : vrai
        modifie         : faux
        modifdate       :              0
        modifusager     :  ~

        ls incoherent   :
        ls perime       :
        ls verrouille   :
        ls incomplet    :

        liste des acces inverses        :
              >ADL>accesbase>accescom.man
              >ADL>accesbase>creer.man
        liste des familles externes     :
              >ADL>accesbase>b>rescatal>hist>lirehist lirehist
              >ADL>accesbase>b>verif>nom nom
              >ADL>accesbase>b>structure>struct struct

        liste des familles filles       :
              delta
              service
              execut
              verif
              rescatal
              structure
              esp

        liste des interfaces associes :
              i1
```

b) manuel du corps b-i1-delta2:

```
manuel   b_i1_delta2
      def

      sel

      attr
           delta = mixte
           hist = oui

      util   = vrai

end
informations en consultation uniquement:

     nom complet    :  >ADL>accesbase>b_i1_delta2.man
     type           :  manuel de corps
     nom universel : nr4
     niveau         :            3
     vide           : faux
     modifie        : faux
     modifdate      :            0
     modifusager    :  -

     ls incoherent :
     ls perime     :
     ls verrouille :
     ls incomplet  :

     existance de fichiers              : vrai

     liste de dependance                :

           >ADL>accesbase>b>delta_i2  niveau :          4
           >ADL>accesbase>b>service_i1  niveau :          8
           >ADL>accesbase>b>execut_i1  niveau :          4
           >ADL>accesbase>b>verif_i1  niveau :          7
           >ADL>accesbase>b>rescatal_i2  niveau :          4
           >ADL>accesbase>b>structure_i1  niveau :          4
           >ADL>accesbase>b>esp_i1  niveau :          9
           >ADL>accesbase>b>rescatal>hist>lirehist_i1  niveau :
           >ADL>accesbase>b>verif>nom_i1  niveau :          8
           >ADL>accesbase>b>structure>struct_i1  niveau :          7
     liste des realisations englobantes:

           >ADL>accesbase_i1_Res  niveau :          1
           >ADL>accesbase>b_i1_Ressai  niveau :          3
           >ADL>accesbase_i1_Rdelta2  niveau :          1
           >ADL>accesbase_i1_RN  niveau :          1
           >ADL>accesbase>b_i1_Rhist  niveau :          3
           >ADL>accesbase>b_i1_RS  niveau :          3
           >ADL>accesbase>b_i1_RN  niveau :          3
     liste des codes objets associes    :

           multics
     numeros des revisions existantes  :
           02
           01
           00

     liste des contraintes héritées     :
```

ANNEXE C.

Exemples de visualisation d'historiques.

Nous présentons ici quelques exemples de visualisation d'historiques. Ils ont entre autres servi de jeux de test lors de la validation du travail.

```
HISTORIQUE DE >ADL>accesbase>b>rescatal>hist>f3_i1_v1
v1.01                    CREE LE   20_12_83..1008   PAR Castaigne.ADL
-------------------------------------------------------------------------
v1.01                    MODIFIE LE 20_12_83..1012   PAR Castaigne.ADL
-------------------------------------------------------------------------
v1.02                    MODIFIE LE 20_12_83..1014   PAR Castaigne.ADL
-------------------------------------------------------------------------
v1.01                    MODIFIE LE 20_12_83..1028   PAR Castaigne.ADL
CREANT >ADL>accesbase>b>rescatal>hist>f3_i1_v2.text.01
-------------------------------------------------------------------------
```

HISTORIQUE DE >ADL>accesbase>b>rescatal>hist>f3_i1_v2

v2.01 CREE LE 20_12_83..1028 PAR Castaigne.ADL

A PARTIR DE >ADL>accesbase>b>rescatal>hist>f3_i1_v1.text.01

---

v2.02 CREE LE 20_12_83..1036 PAR Castaigne.ADL

A PARTIR DE >ADL>accesbase>b>rescatal>hist>f3_i1_v3.text.01

HISTORIQUE DE >ADL>accesbase>b>rescatal>hist>f3_i1_v3

v3.01 CREE LE 20_12_83..1033 PAR Castaigne.ADL

---

v3.01 MODIFIE LE 20_12_83..1036 PAR Castaigne.ADL

CREANT >ADL>accesbase>b>rescatal>hist>f3_i1_v2.text.02

HISTORIQUE DE >ADL>accesbase>b>rescatal>hist>f3_i1_v4

v4.01 CREE LE 20_12_83..1351 PAR Castaigne.ADL

---

v4.01 MODIFIE LE 20_12_83..1355 PAR Castaigne.ADL

---

v4.02 MODIFIE LE 20_12_83..1400 PAR Castaigne.ADL

---

v4.02 MODIFIE LE 20_12_83..1412 PAR Castaigne.ADL

---

v4.04 MODIFIE LE 20_12_83..1415 PAR Castaigne.ADL

---

v4.04 MODIFIE LE 20_12_83..1417 PAR Castaigne.ADL

CREANT >ADL>accesbase>b>rescatal>hist>f3_i1_v5.text.01

HISTORIQUE DE >ADL>accesbase>b>rescatal>hist>f3_i1_v5

v5.01 CREE LE 20_12_83..1417 PAR Castaigne.ADL

A PARTIR DE >ADL>accesbase>b>rescatal>hist>f3_i1_v4.text.04

---

v5.01 MODIFIE LE 20_12_83..1423 PAR Castaigne.ADL

---

v5.02 MODIFIE LE 20_12_83..1427 PAR Castaigne.ADL

---

v5.02 MODIFIE LE 20_12_83..1430 PAR Castaigne.ADL

CREANT >ADL>accesbase>b>rescatal>hist>f3_i1_v6.text.01

HISTORIQUE DE >ADL>accesbase>b>rescatal>hist>f3_i1_v6

v6.01 CREE LE 20_12_83..1430 PAR Castaigne.ADL

A PARTIR DE >ADL>accesbase>b>rescatal>hist>f3_i1_v5.text.02

---

v6.01 MODIFIE LE 20_12_83..1434 PAR Castaigne.ADL

---

v6.01 MODIFIE LE 20_12_83..1439 PAR Castaigne.ADL

---

v6.03 MODIFIE LE 20_12_83..1442 PAR Castaigne.ADL

CREANT >ADL>accesbase>b>rescatal>hist>f3_i1_v7.text.01

---

v6.03 MODIFIE LE 19_01_84..1738 PAR Castaigne.ADL

---

v6.04 MODIFIE LE 19_01_84..1741 PAR Castaigne.ADL

```
      HISTORIQUE  DE >ADL>accesbase>b>rescatal>hist>f3_i1_v7

v7.01                      CREE  LE   20_12_83..1442   PAR Castaigne.ADL

A PARTIR DE >ADL>accesbase>b>rescatal>hist>f3_i1_v6.text.03
-------------------------------------------------------------------------

v7.01                      MODIFIE LE 21_12_83..1445   PAR Castaigne.ADL

-------------------------------------------------------------------------
```

L'exemple de la page suivante présente la visualisation de la provenance de la version v7.

PROVENANCE COMPLETE DE >ADL>accesbase>b>rescatal>hist>f3_i1_v7

v4.01 CREE LE 20_12_83..1351 PAR Castaigne.ADL

---

v4.01 MODIFIE LE 20_12_83..1355 PAR Castaigne.ADL

---

v4.02 MODIFIE LE 20_12_83..1400 PAR Castaigne.ADL

---

v4.02 MODIFIE LE 20_12_83..1412 PAR Castaigne.ADL

---

v4.04 MODIFIE LE 20_12_83..1415 PAR Castaigne.ADL

---

v4.04 MODIFIE LE 20_12_83..1417 PAR Castaigne.ADL

CREANT >ADL>accesbase>b>rescatal>hist>f3_i1_v5.text.01

---

v5.01 CREE LE 20_12_83..1417 PAR Castaigne.ADL

A PARTIR DE >ADL>accesbase>b>rescatal>hist>f3_i1_v4.text.04

---

v5.01 MODIFIE LE 20_12_83..1423 PAR Castaigne.ADL

---

v5.02 MODIFIE LE 20_12_83..1427 PAR Castaigne.ADL

---

v5.02 MODIFIE LE 20_12_83..1430 PAR Castaigne.ADL

CREANT >ADL>accesbase>b>rescatal>hist>f3_i1_v6.text.01

---

v6.01 CREE LE 20_12_83..1430 PAR Castaigne.ADL

A PARTIR DE >ADL>accesbase>b>rescatal>hist>f3_i1_v5.text.02

---

v6.01 MODIFIE LE 20_12_83..1434 PAR Castaigne.ADL

---

v6.01 MODIFIE LE 20_12_83..1439 PAR Castaigne.ADL

---

v6.03 MODIFIE LE 20_12_83..1442 PAR Castaigne.ADL

```
CREANT >ADL>accesbase>b>rescatal>hist>f3_i1_v7.text.01
-------------------------------------------------------------------------------

v7.01                    CREE  LE   20_12_83..1442   PAR Castaigne.ADL

A PARTIR DE >ADL>accesbase>b>rescatal>hist>f3_i1_v6.text.03
-------------------------------------------------------------------------------

v7.01                    MODIFIE LE 20_12_83..1445   PAR Castaigne.ADL

-------------------------------------------------------------------------------
```